COMPILER ANALYSIS AND OPTIMIZATION OF MEMORY MANAGEMENT IN MODERN PROCESSORS

A Dissertation
Presented to
The Academic Faculty

By

Prithayan Barua

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology

May 2021

COMPILER ANALYSIS AND OPTIMIZATION OF MEMORY MANAGEMENT IN MODERN PROCESSORS

Approved by:

Dr. Vivek Sarkar, Advisor
School of Computer Science
Georgia Institute of Technology

Dr. Hyesoon Kim
School of Computer Science
Georgia Institute of Technology

Dr. Rich Vuduc
School of Computational Science and Engineering
Georgia Institute of Technology

Dr. Santosh Pande
School of Computer Science
Georgia Institute of Technology

Dr. Tom Conte
School of Computer Science
Georgia Institute of Technology

Date Approved: January 14, 2020

ACKNOWLEDGEMENTS

First and foremost, I would like to extend my deepest gratitude to my advisor Dr. Vivek Sarkar without whom this work would not have been possible. His unwavering support, mentorship, and guidance helped me grow as a graduate student. I thank him for the constant motivation and giving me the opportunity to do very interesting research with the Habanero Extreme Scale Software Research Group.

I want to thank Prof. Hyesoon Kim, Prof. Rich Vuduc, Prof. Santosh Pande, and Prof. Tom Conte for agreeing to be part of my thesis committee. I am grateful for their time and all the feedback and suggestions that significantly improved my thesis.

I am grateful to all my collaborators, Jun Shirako, Jisheng Zhao, Lechen Yu, Prasanth Chatarasi, Sana Damani, Girish Mururu, Chris Porter from Georgia Tech, Bardia Mahjour, Ettore Tiotto, Jeeva Paudel, Wang Chen, Whitney Tsang from IBM, Hongbo Rong from Intel, Nitish Srivastava from Cornell. Thanks for the immense help, support, and guiding me in my research work. Without their collaboration, this thesis would not have been possible. I am also thankful to Johannes Doerfert from Argonne National Lab for the various insights and exciting discussions. I also want to thank Xinmin Tian for the rich experience I gained at my internship at Intel and Derrick Aguren, Vignesh Adhinarayanan and Sreejith Menon for their mentorship during my internship at AMD.

In addition to all the collaborators mentioned above, I want to acknowledge the Habanero Extreme Scale Software Research Group and CRNCH Research Center (Jeffery Young) for the research interactions in my Ph.D. journey. Thanks are also due to all my friends, family and mentors who kept me motivated and supported me throughout my Ph.D. Journey. I want to acknowledge Professor Y.N Srikant, who advised me during my Masters at Indian Institute of Science and motivated me to continue my graduate studies.

Finally the most important, I would like to conclude by acknowledging that I am indebted to my mother Pamela Barua, my father Pulak Barua, and my sister Shreya Barua

for their unconditional love. Without their encouragement and support, I could not have reached this far.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Modern processors, such as CPUs and GPUs have continued to deliver higher performance with increasing microarchitecture innovations in recent years. However, it is non-trivial for application developers to achieve anywhere close to peak performance on these platforms. Memory is the most common system performance bottleneck because main memory performance has not kept up with the pace of improvement in processor performance.

Caches generally serve as a first level of the memory hierarchy to hide the longer main memory access latency, which can be up to 100$\times$ slower than caches. But on-chip caches can take up a lot of silicon area, consume significant amounts of processor power, and are also one of the most expensive per byte of memory. Hence, efficient utilization of data in the cache memory is always one of the most crucial performance optimizations for application developers. This is the first problem we address in our thesis. We develop a static compiler analysis to model the data cache usage of a given loop nest. Then we use the cache analysis in a cost model to guide the unroll-jam loop transformation with the objective of maximizing reuse of the data in cache. This compiler optimization can help developers improve the performance of their applications on modern CPUs.

When considering main memory (which is placed above several levels of cache in the memory hierarchy), we see that main memory access latency has improved $< 2\times$ over the last two decades, but the bandwidth has continued to improve by over 100$\times$. Modern graphics processing units (GPUs) exploit this as a distinguishing feature to sustain thousands of concurrent threads. The high-bandwidth GPU memory hides the long memory access latency by issuing multiple memory accesses from different threads in parallel. It can be non-trivial for application developers to efficiently utilize this high memory bandwidth, especially when porting existing applications from multicore CPUs to GPUs. This thesis proposes a static analytical model for the GPU memory bandwidth utilization of a

kernel. We then use the analysis to introduce a new cost model to guide a thread coarsening transformation to improve bandwidth utilization.

In the memory hierarchy for accelerator devices like GPUs, the Host (CPU) memory is the next level in the organization. Any data required by a kernel executing on the GPU has to be copied from the CPU main memory to the device memory. This memory copy is one of the slowest operations when offloading kernels to accelerator devices, and can dominate the execution time of many GPU applications. Since device memory usually persists across multiple kernel instances, so there is an opportunity to reuse the data in the device memory across multiple kernel executions. As a final part of our thesis, we develop an intermediate representation to model host-device memory copy operations. Then we use it first to design an analysis to detect incorrect usage of host-device memory copy in OpenMP applications. Then we develop an optimization to remove redundant host-device memory copies. Our compiler tools can improve developer productivity and deliver high performance by automatically managing the GPU memory hierarchy.

"Given the increasing disparity between processor and memory performance and that memory will continue to remain the critical system performance bottleneck in modern processors, our thesis is that advances in static compiler analysis are critical to improve programmability and enable compiler optimizations for maximizing data reuse and memory bandwidth utilization and minimizing redundant memory accesses in modern processors."

CHAPTER 1

INTRODUCTION

1.1 Background

With advances in modern computer architecture, processor performance has continued to increase with multicore CPUs and graphics processing units (GPUs). Modern processors implement several advanced optimizations like branch prediction or special-function-units in the microarchitecture. Such features can improve the performance of existing applications without requiring changes in software. However, despite these advances in modern high-performance processors, it is still difficult for typical applications to achieve anywhere close to peak performance. The main reason is that memory is the most common system performance bottleneck, and memory performance has not kept up with the pace of improvement in processor performance.

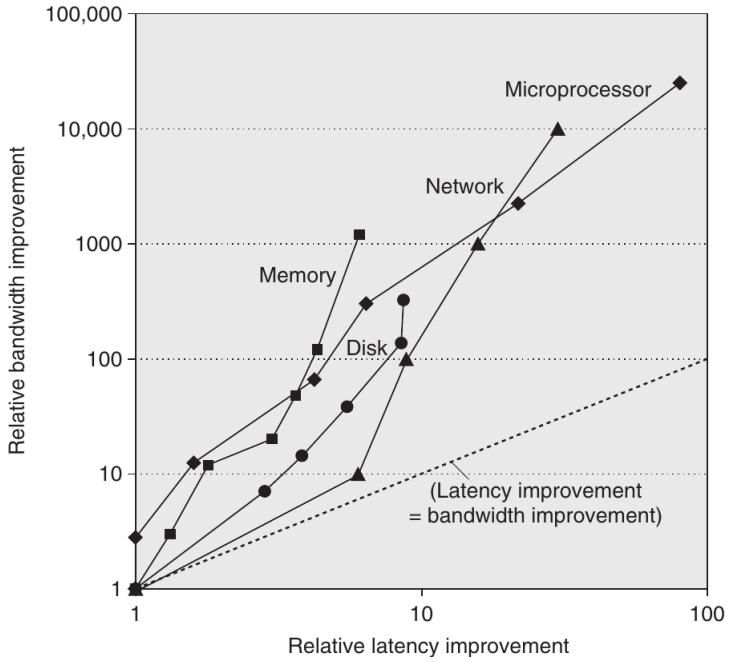


Figure 1.1: Figure 1.9 from [1], Log-log plot of bandwidth and latency

Figure 1.1 [1] plots the improvement in bandwidth and latency for technology milestones for microprocessors, memory, networks and disks. Processors have seen the greatest gains of about 25,000$\times$ in bandwidth and 80$\times$ in latency. Memory lags much behind with about 1000$\times$ improvement in bandwidth but only 8$\times$ in latency.

The memory hierarchy is a solution to an application's demand for unlimited amounts of fast memory. It exploits the general principles of locality and cost-performance trade-off. Spatial locality refers to using data elements stored close together, and temporal data locality refers to reusing a data element over time. Further, smaller sized memory is usually faster for a given implementation technology, and higher performance memory is more expensive.

The memory hierarchy comprises several levels, starting from the smallest size memory that is closest to the processor. Each level of memory that is farther from the processor is usually larger, slower, and less expensive per byte. The first few storage levels are usually on-chip and consist of processor registers and caches, followed by the off-chip main memory consisting of RAM.

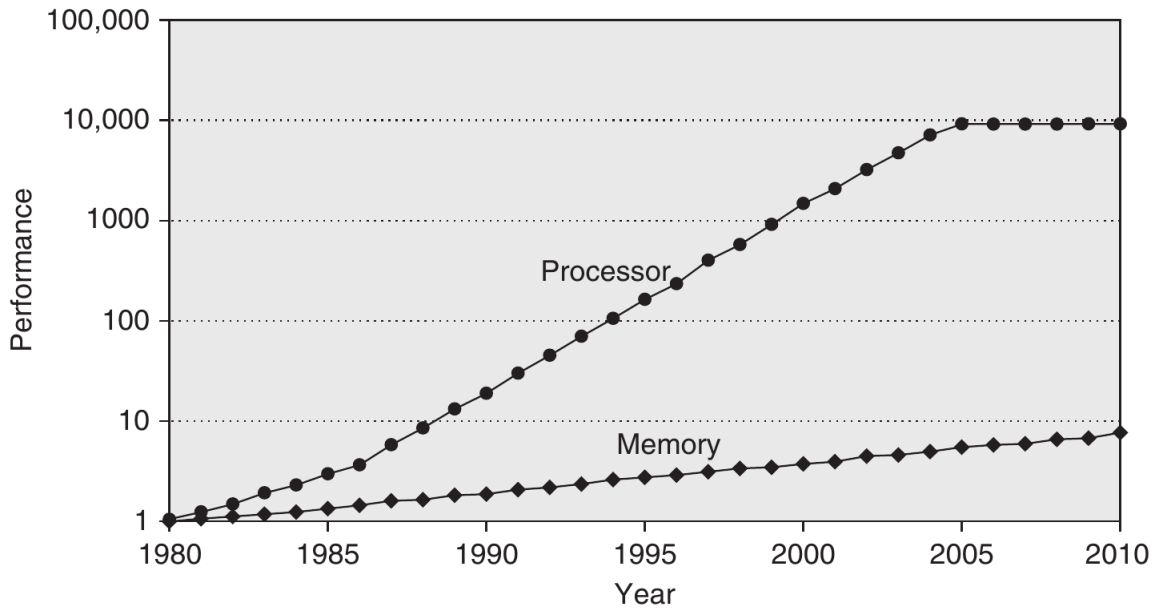


Figure 1.2: Figure 2.2 from [1], gap in performance measured as the difference in the time between processor memory requests and the latency of DRAM access.

The memory hierarchy's significance has continued to increase with the growing disparity between processor and memory performance. The term memory wall [2] was coined to describe the situation, where the improvements in processor performance are eventually masked by the relative slow improvements to DRAM speed. Eventually, the total time spent waiting for a memory fetch from DRAM will dominate the total execution time even if the relative number of memory references is small.

Figure 1.2 from [1] plots the increase in memory requests from a single processor per second on average to the increase in DRAM accesses per second.

On-chip cache memory is about 100 times faster than off-chip DRAM, but it takes up a lot of on-chip silicon area and a significant percentage of processor power. Hence it is critical to make efficient use of the limited cache memory. Traditionally it's the job of expert programmers to understand the underlying memory hierarchy and implement memory management optimizations. There has also been a lot of research in compilers to model the data cache memory [3, 4, 5, 6, 7, 8, 9] and use the analysis to implement transformations that improve reuse in the cache [10, 11, 12, 13]. We discuss our contribution to compiler optimizations for cache memory reuse and related work in chapter 2.

Figure 1.3 illustrates the historical scaling trends of a DRAM chip[14]. While the capacity and bandwidth of DRAM-based main memory have made rapid progress by improving 128× and 20× over the past two decades, latency has only improved by 1.3×. This

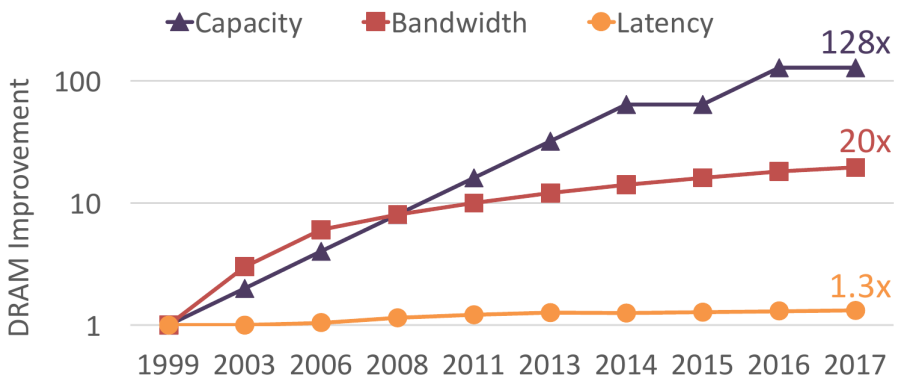


Figure 1.3: DRAM scaling trend over time, figure from [14]

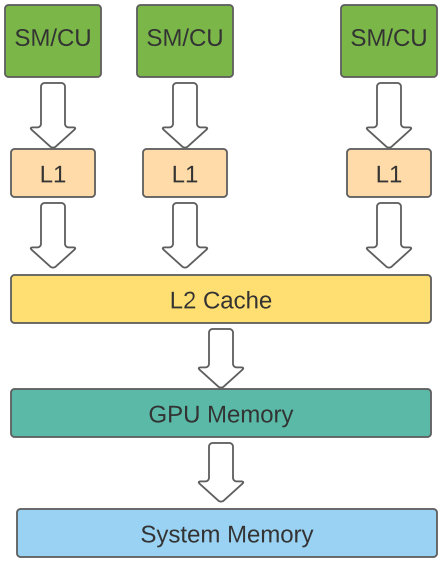


Figure 1.4: GPU memory hierarchy, SM=Streaming multiprocessor(NVIDIA), CU=Compute unit(AMD)

shows that DRAM latency continues to remain a significant system performance bottleneck for most modern applications.

Modern processors like GPUs exploit this trend by increasing the available memory bandwidth and issue multiple memory requests concurrently to hide the long memory latency.

GPU memory hierarchy Figure 1.4 shows a general GPU memory hierarchy. The L1 cache is usually an on-chip cache, private to each compute-unit/streaming-multiprocessor, while L2 can be a shared cache. The GPU memory is a very high bandwidth onboard GPU memory, and System memory is the host memory. It is critical to maximize the GPU memory bandwidth utilization to achieve peak GPU performance. Furthermore, the memory copy between host system memory and GPU memory can be very slow, and this adds another challenge to GPU memory management. Since the GPU memory can be persistent across multiple kernel launches, it is crucial to reuse data across multiple kernel executions. Programmers typically spend a lot of effort manually implementing such memory optimizations [15, 16, 17, 18, 19], but can also introduce errors when doing so. There

has also been a lot of research in compilers to make it easier to program GPUs and automate memory management [20, 21, 22, 23]. We discuss such compiler optimizations and programmability tools in Chapters 3, 4 and 5.

1.2 Thesis Statement

"Given the increasing disparity between processor and memory performance and that memory will continue to remain the critical system performance bottleneck in modern processors, our thesis is that advances in static compiler analysis are critical to improve programmability and enable compiler optimizations for maximizing data reuse and memory bandwidth utilization and minimizing redundant memory accesses in modern processors."

1.3 Contributions

1.3.1 Static cache modeling to optimize data reuse via loop unrolling

We propose a static analysis to model L1 cache misses and use the loop unroll-jam transformation to maximize cache line reuse at L1. It is developed as a new static analysis pass called "OptiMemReuse" in LLVM. We use it to guide the selection of unroll-jam transformations with the objective of reducing the memory cost of a loop nest while also taking register pressure into account. To do so, we had to extend past work on cache models to also consider outer (non-innermost) loops, as well as constraints such as limited set associativity that are important when modeling real hardware. OptiMemReuse can be used to estimate the cache misses that would occur after performing the unroll- jam transformation for a given unroll configuration, without actually performing the transformation.

We evaluated OptiMemReuse on three hardware platforms – IBM POWER9, AMD Ryzen 9, and Intel(R) Xeon(R) Gold – by implementing it in LLVM and comparing the effect of unroll-jam driven by the OptiMemReuse cost model with the baseline model currently used in LLVM. The results obtained across all 30 PolyBench benchmarks are very

encouraging. There was no degradation of more than 2% across all the platforms. The performance improvements obtained by the use of OptiMemReuse were observed to be up to 2.57× for IBM POWER9 (geometric mean of 1.21×), up to 4.62× for AMD Ryzen (geometric mean of 1.33×), and up to 5.26× for Intel Xeon (geometric mean of 1.16×).

1.3.2 Static GPU memory bandwidth modeling to improve bandwidth utilization via thread coarsening transformation

We observed that traditional applications designed for latency optimized out-of-order pipelined CPUs do not exploit the throughput optimized in-order pipelined GPU architecture efficiently. We develop a model to estimate the memory throughput of a given application [24]. We can use this analysis to statically identify kernels that have low GPU memory bandwidth utilization. Then we use the loop interleaving transformation to improve the memory bandwidth utilization of a given kernel. We developed a heuristic to estimate the optimal loop interleave factor, and implemented it in the OpenARC compiler for OpenACC.

Directive-based programming models like OpenACC provide a higher level abstraction and low overhead approach of porting existing applications to GPGPUs and other heterogeneous HPC hardware, relative to GPU-specific programming models like CUDA. Such programming models increase the design space exploration possible at the compiler level to exploit specific features of different architectures.

We evaluated our approach on over 216 kernels to achieve a geometric mean speedup of 1.32×. Our compiler optimization aims to provide a desirable balance across performance, portability and productivity with this approach.

1.3.3 Static modeling of Host-GPU memory movement for debugging and optimization

We observed that the cost of high volume data movement between the host and the accelerators(GPU) is a critical bottleneck both in terms of application performance and developer

productivity. Memory management is usually a manual task performed tediously by expert programmers. We designed a new intermediate representation called Location-aware Array SSA(LASSA) to model memory copy operations between the host and accelerators[25]. We chose OpenMP as the underlying parallel programming model and implemented our analysis framework in the LLVM toolchain.

Using the LASSA analysis, we developed **OMPSan**[26] (OpenMP Sanitizer) – a static analysis-based tool that helps developers detect bugs from incorrect usage of the `map` clause while trying to optimize host-device data transfers, and also suggests potential fixes for the bugs. OmpSan utilizes a data flow analysis that validates if the def-use information of the array variables are respected by the mapping constructs in the OpenMP program. We evaluated **OmpSan** over some standard benchmarks and also show its effectiveness by detecting commonly reported bugs.

Based on the LASSA representation, we also introduce an optimization framework that casts the problem of detection and removal of redundant data movements into a partial redundancy elimination (PRE) problem and applies the lazy code motion technique to optimize these data movements. We evaluated it with ten benchmarks and obtained a geometric speedup of 2.3×, and reduced on average 50% of the total bytes transferred between the host and GPU.

CHAPTER 2
STATIC CACHE MODELING TO OPTIMIZE DATA REUSE VIA LOOP UNROLLING

2.1 Introduction

With advances in microarchitectural features, modern high-performance processors implement several advanced optimizations in hardware. Many such features were traditionally the objective of some compiler transformations. For example, the loop unroll transformation is traditionally used for reducing branch overhead and increasing instruction-level parallelism of a loop nest. But modern processors like the IBM POWER9 can speculatively execute branches overlapped with other loop instructions, making the branch effectively execute in zero cycles, so loop unrolling with an objective of reducing branch overhead may not be relevant in many circumstances. For example, we have observed that the average branch misprediction rate on the IBM POWER9 architecture to be $< 0.1\%$ for the entire PolyBench[27] benchmark suite, which is not surprising given the hardware capabilities of POWER9 and other recent processors, and the fact that PolyBench contains loop intensive benchmarks.

While arithmetic and branch prediction units have advanced significantly in recent years, data-movement still remains a fundamental bottleneck in many applications as the gap between memory latency continues to increase. Therefore, any optimization that can improve data locality or memory reuse can have a significant impact on actual performance.

In our experiments, we have noticed that the loop unroll and jam transformation (referred to as unroll-jam in this chapter) has significant implications on data locality and memory reuse. The motivation for our work comes from the observation that loop unrolling heuristics used by a state-of-the-art production compiler like LLVM do not take

into account the memory cost of the transformation. To quantify the memory cost of the unroll-jam transformation, we need to consider several nuances of cache modeling at compile time. The existing LLVM cache modeling analysis is adapted from [28], and does not consider reuse across outer loops and also does not take cache associativity into account. We will show in subsection 2.2.3, that both of these factors are critical in understanding the performance implications of the unroll-jam transformation.

In this work, we propose OptiMemReuse, an analysis to guide the loop unroll-jam transformation with the objective of reducing the memory cost of a loop nest. OptiMemReuse models a set-associative L1 data cache and estimates the total cache misses of any given loop nest. The analysis can account for reuse opportunities, to estimate the cache misses after performing the loop unroll-jam transformation by a given factor without actually performing the transformation. It prescribes the unroll factors for nested loops that can maximize data locality and memory reuse.

OptiMemReuse is an analysis pass that estimates the benefits of a loop transformation and guides the compiler in when best to perform the transformation.

While we demonstrate the effectiveness of the memory cost model on the unroll-jam transformation in this chapter, we believe that it is applicable to other transformations as well.

While most recent work on static modeling of cache behavior [5, 29] focuses on the accuracy of the model for affine programs, OptiMemReuse is a fast and effective static cost model designed to compare the relative benefits of different loop transformations in terms of memory cost.

In summary, we make the following contributions,

- An efficient modeling of set-associative caches to estimate total cache misses of a loop nest and modeling memory reuse across iterations of loops at arbitrary depth in a loop nest.

- Integrate the memory model into the loop unroll-jam cost model, to estimate the

memory cost of the loop unroll-jam transformation without actually performing the transformation.

- Our analysis includes a register pressure estimation algorithm to consider the additional memory accesses due to register spills. We model the effect of a loop transformation like unroll-jam on register pressure without actually performing the transformation.

2.2 Background

2.2.1 Hardware Caches

To simplify our analysis we assume that any memory access instruction can lead to the following events in a set-associative L1 data cache.

- Load/Store instruction issues a virtual address, which gets translated to the physical address.
- The physical address is mapped to a corresponding L1 cache set.
- If the cache line containing the address is not present, it is requested from the L2 cache.
- The fetched cache line from L2 is placed in the corresponding cache set in L1.
- If the cache set was full then the least recently used cache line is evicted.
- We assume a write-back cache update policy.

Statically modeling all the above events accurately is not feasible in most cases. For example, a dynamically allocated memory address is not known at compile time. So, the static analysis makes several simplifying and conservative assumptions to make it possible to model the complex cache behavior.

Figure 2.1 outlines the cache behavior of a simple loop with one-dimensional array access.

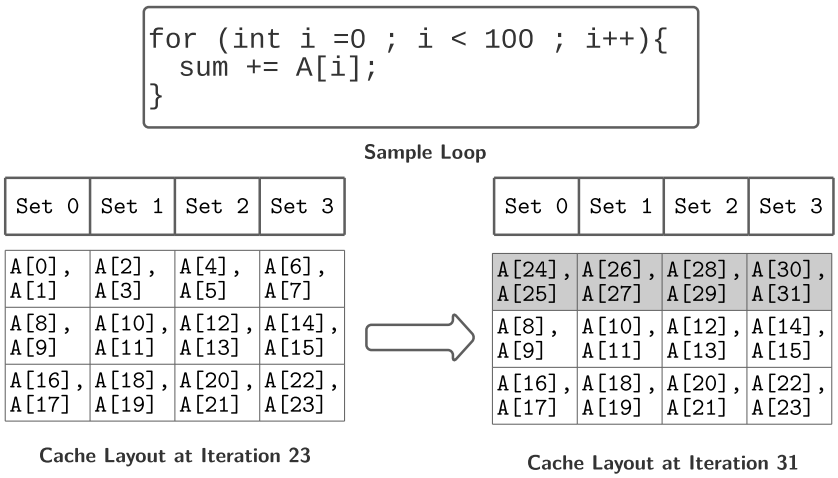


Figure 2.1: Illustrative example of 3-way set associative cache with 4 cache sets

In this example we consider a 3-Way set-associative cache with a total of 4 cache sets, and every cache line consists of 2 array elements (ignore the element size). In Figure 2.1 we assume the base address $A[0]$ maps to Set-0. Then access to $A[1]$ hits the cache line corresponding to $A[0]$ in set-0. Then the next cache line corresponding to $A[2], A[3]$ are mapped to set-1, and so on. In this example every other access is a cache miss. It also illustrates how the array elements are mapped to different cache sets. The cache line corresponding to $A[8], A[9]$ is wrapped around and mapped to cache set-0. Typically the least significant bits of a memory address are used to determine the cache set. For the purpose of statically modeling the set associativity we can assume the cache layout as illustrated in the Figure 2.1. Now consider the access to $A[24]$, which maps back to set-0, which is already full at iteration 24. So, the least recently used cache line, $(A[0], A[1])$ is evicted. And this continues for the following iterations as illustrated in the Figure 2.1.

2.2.2 Unroll-Jam Loop Transformation

Unroll-Jam is the combined operation of loop unrolling and jamming [30]. For example if we apply the unroll-jam transformation on Listing 2.1 by a factor of (2,1) then the result is Listing 2.2. The jamming must maintain the order of data dependencies. If it reverses the execution order of array references then it becomes illegal[31, 32].

2.2.3 Motivation: Optimal Unroll-Jam Factor

In this section we show an example kernel from the Polybench benchmark [27] to illustrate the cache reuse opportunities exposed by the unroll-jam transformation, and the significance of modeling the cache associativity to reason about its performance implications. The Listing 2.1 shows a simplified version of the Bicg kernel and Listing 2.2 shows the corresponding loop nest after the unroll-jam. The loop i was unrolled by a factor of 2, while the loop j was unrolled by a factor of 1.

```
1 for (i = 0; i < N; i++) {
2   for (j = 0; j < M; j++) {
3     s[j] = s[j] + r[i] * A[i][j];
4   }}
```

Listing 2.1: Bicg Kernel

```
1 // Unroll-Jam by (2,1)
2 for (i = 0; i < N; i+=2) {
3   for (j = 0; j < M; j++) {
4     s[j] = s[j] + r[i] * A[i][j];
5     s[j] = s[j] + r[i+1] * A[i+1][j];
6   }}
```

Listing 2.2: Bicg Kernel after unroll-jam by (2,1)

We experimented on IBM POWER9, to evaluate the performance implications of different unroll factors, as shown in Figure 2.2. The graph shows that the runtime keeps decreasing up to the unroll-jam factor of (4,8), but increases by almost 20% from (8,1). We also plot the L1 data cache-miss for each experiment and show a strong correlation between the reduction/increase in L1 misses and the obtained speedup/slowdown. The

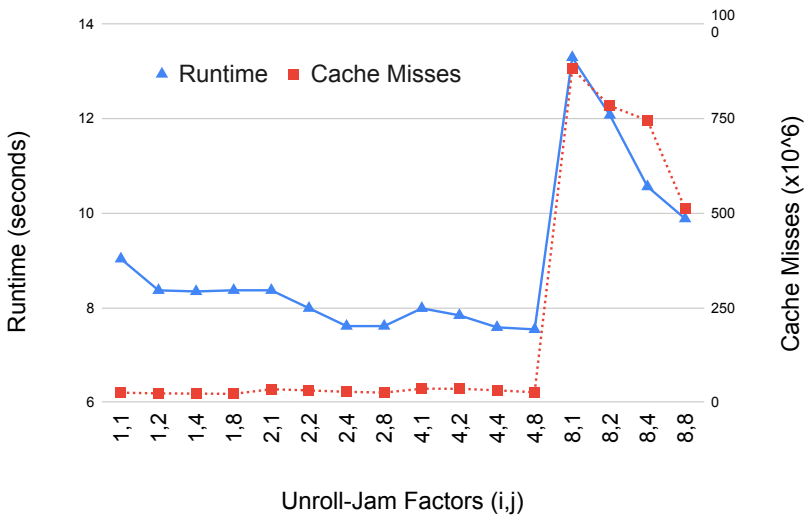


Figure 2.2: Bicg Application Runtime vs L1 Data Cache misses

sudden increase in cache-misses after unroll-jam of outer-loop by a factor of 8 is because of cache associativity. On the IBM POWER9, the L1 is an 8-way set-associative cache. Now, consider the accesses to array A in the loop body after unroll-jam by a factor of (8,1): $A[i][j]$, $A[i+1][j]$, $A[i+2][j]$, ..., $A[i+7][j]$. All the above accesses map to the same cache set and increase conflict misses. This causes cache thrashing and precludes even the cache line reuse across the iterations of the j loop.

2.3 Overview

Figure 2.3 shows an overview of our tool OptiMemReuse. Note that every analysis in the tool estimates certain property of a given loop body after the unroll-jam transformation by a given factor, without actually performing the transformation. Given a loop nest of n loops, step 2 iterates over all possible unroll factors. The replace space is constrained by the instruction cache size requirement(step 4) and L1 data cache conflict misses due to set-associativity(step 8). Step 3 estimates the size of the loop body for the given unroll factors, which is used to constrain the replace space based on the available instruction cache size. Step 5 estimates the register pressure in the loop body for the given unroll factors. Based on

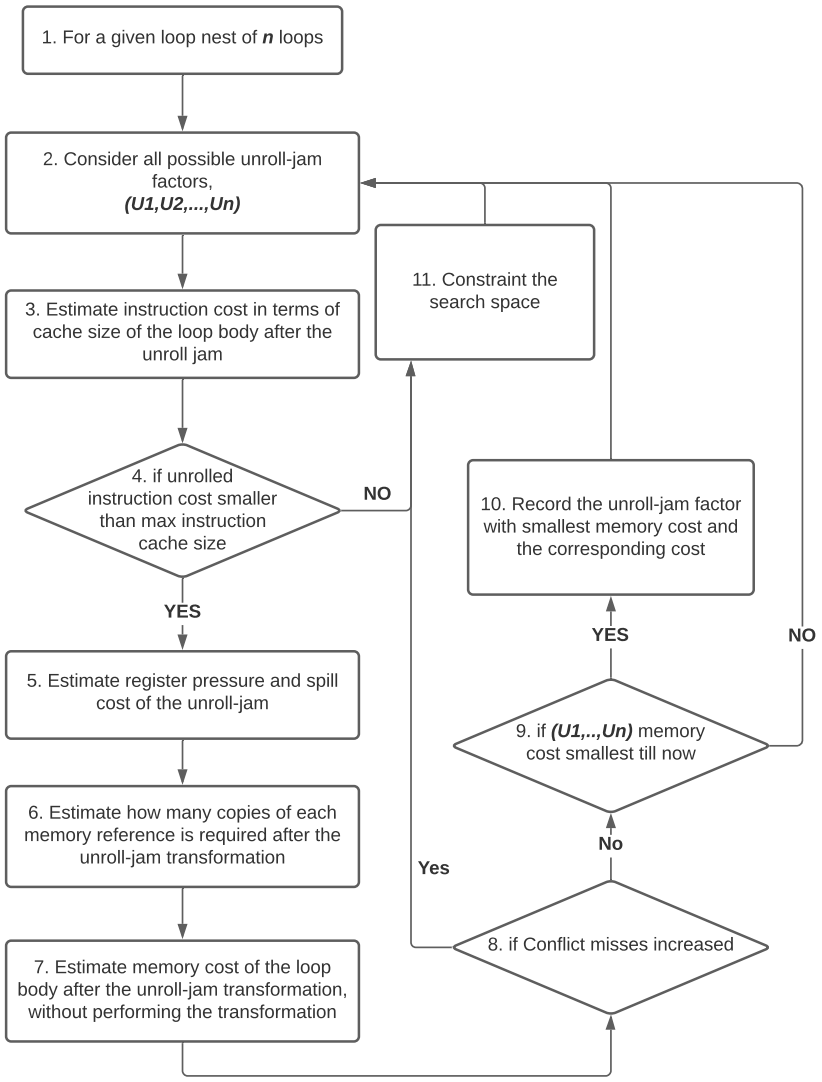


Figure 2.3: Overview of OptiMemReuse, search for the best unroll-jam factor that minimizes memory cost

the available registers for the given architecture, total register spills are estimated, and this is added to the memory cost of the loop body. Subsection 2.4.5 discusses step 5 in detail. Step 6 considers any reuse opportunities to estimate the total copies of each instruction required for the given unroll factors, it is explained in subsection 2.4.4. Finally, step 7 (subsection 2.4.2)estimates the memory cost based on steps 5 and 6. The total conflict misses keep increasing with larger unroll factors; hence it is used in step 8 to constraint the replace space. Lastly, steps 9 and 10 record the unroll factors that result in minimum memory cost.

2.4 Data Cache Modeling

In this section, we estimate the memory cost of a loop nest. We use cache misses as a primary estimate of the cost of a loop nest. We assume that any transformation that can reduce the memory cost of a loop nest should be beneficial in general.

2.4.1 Cache Misses

There are three kinds of cache misses[33], that our analysis should model,

1. *compulsory misses* happen if a program accesses a cache line for the first time,
2. *capacity misses* happen if a program accesses too many distinct cache lines before accessing a cache line again
3. *conflict misses* happen if a program accesses too many distinct cache lines that map to the same set of an associative cache before accessing the cache line again.

To simplify the discussions, we consider only compulsory and capacity misses in this section, assuming a fully associative cache. But in the following subsection 2.4.3 we extend our model to handle set associative caches also.

Cache Line Size, Cache Capacity: We use the term CLS to denote the cache line size. CLS is defined in bytes most generally, but for illustrative purposes we will assume that

it equals the number of array elements for the examples discussed in this chapter. Also $CacheCpacity$ denotes the size of the cache, also in terms of array elements.

Our cache modeling analysis is derived from [34], and we extend it by considering the cost of non-innermost loops (which is critical for unroll-jam) and also considering capacity and conflict misses. Hence we use most of the notations and conventions from [34] in our description below.

We use loop depth to denote the nesting level of any loop in a loop nest, where the outermost loop has a depth of 1.

For all the loops in a given loop nest, we perform the following steps, starting from the innermost,

1. Estimate the total cache lines required per iteration of the loop, considering intra-iteration cache line reuses

2. Estimate the total cache lines required over all iterations of the loop, considering inter-iteration reuses

3. Repeat the above steps for the immediately outer loop

For the innermost loop, the cost of one iteration is simply the cost of executing the memory references in the loop body once. The cost of one iteration of an intermediate loop at depth d is equal to the cost of the immediately inner loop at depth $d+1$, that is why some of the terms we use in this section will be defined recursively.

2.4.2 Cache Line Reuse Analysis

This section estimates the total number of cache line fetches required by a given loop nest.

Loop: We use the term L_d to represent a loop, which has three properties, the total number of iterations denoted by $trip$, the step increment denoted by $step$, and loop depth, d.

Reference: Any N dimensional array reference Ref has two properties. A base pointer denoted by, $BasePointer(Ref)$ and a sequence of subscripts $f_1(Ref)$ to $f_N(Ref)$, innermost to outermost. For example, $A[i][j+2][k*10]$, has a base pointer of A, $f_1 = (k*10)$, $f_2 = (j+2)$ and $f_3 = (i)$.

One of the primary components of cache analysis is to estimate when two references can request to fetch the same cache line. We consider two cache line reuse scenarios: within a single iteration (intra-iteration) and across all iterations of a loop (inter-iterations). The analysis is performed for each loop in the loop nest, starting from the innermost.

Two References Ref_1, Ref_2 exhibit intra-iteration reuse if,

- $BasePointer(Ref_1) == BasePointer(Ref_2)$ and
- $f_i(Ref_1) == f_i(Ref_2), \forall i \neq 1$ (All subscripts except innermost are exactly same) and
- $|f_1(Ref_1) - f_1(Ref_2)| < CLS$ (Innermost subscript difference within CLS)

Two References Ref_1, Ref_2 refer to the same cache line across iterations of L_d, where d is the depth of the loop, if $\exists\ Ref_1\ \vec{\delta}\ Ref_2$, where $\vec{\delta}$ is the data dependence [35, 36] expressed as a distance vector from outermost loop to innermost loop, $\vec{\delta} = \{\delta_1, ..., \delta_k\}$, where k is the depth of the innermost loop, and the following conditions hold,

- $\vec{\delta}$ is loop-independent, or
- – $\delta_x = 0, \forall x < d$, and
  – δ_d is a constant

Also, note that we consider all the loops in a loop nest starting from the innermost loop, hence all kinds of dependence distances are handled here.

The above conditions can estimate when two references fetch the same cache line. It is similar to the analysis developed in [34], and we extend it by considering the following conditions to decide if the two references result in a cache hit,

1. For the intra-iteration reuse, the total cache lines required to execute a single iteration must be less than the cache capacity

2. For the inter-iteration reuse across K iterations, the total cache lines fetched for K iterations must be less than the cache capacity

3. The third condition to consider eviction due to set associativity is discussed in subsection 2.4.3.

For example, references $Ref_1 = A[i][j+1][k]$ and $Ref_2 = A[i][j][k]$, have a distance vector of $(0, 1, 0)$. Assuming i is the outermost loop and k the innermost. Now, Ref_2 of iteration $(j+1)$ can potentially reuse the cache line fetched by Ref_1 in iteration j. Here, the reuse distance is 1 iteration of j loop. Hence for Ref_2 to be a cache-hit in $(j+1)$, all the cache lines fetched in one j iteration shouldn't evict Ref_1. In this example, the reuse across iterations of loop j, doesn't depend on the inner loops' distance, since inner loops are completely executed in one iteration of the outer loop.

Reference Group: A reference group for a loop is the set of all references accessing and reusing the same cache line within the loop.
$RefGroup_{intra}(Ref, L_d)$ denotes the reference group that Ref belongs to in one iteration of loop L_d. Ref_1 and Ref_2 belong to the same reference group in one L_d iteration, if:

1. They access the same cache line in one L_d iteration, and

2. The cache line is not evicted between the two accesses

As an example, references, $A[i][j][k]$ and $A[i][j][k+1]$ exhibit intra-iteration reuse.

$RefGroup(Ref, L_d)$ denotes the reference group that Ref belongs to for the entire loop L_d. Ref_1 and Ref_2 belong to the same reference group with respect to the entire loop L_d, if:

1. They access the same cache line across iterations of L_d

2. The cache line is not evicted between the two accesses

For example, $Ref_2 = A[i][j+4][k+1]$ and $Ref_1 = A[i][j][k]$, can potentially reuse a cache line across 4 iterations of j.

Loop iteration footprint: $LoopFootprint_{iter}(L_d)$ denotes the total cache lines fetched in a single iteration of loop L_d. Which can be estimated as the total number of unique $RefGroup_{intra}(Ref, L_d),\ \forall Ref \in L_d$, that is the total number of intra-loop reference groups, since only one cache line is required for all references in the same reference group. It can be used to estimate if a cache line reuse is possible across K iterations of L_d,
$ReusePossible(K, L_d) =$
$(K * LoopFootprint_{iter}(L_d)) < CacheCapacity.$
Here assuming we have a fully set-associative cache, if K iterations of the loop fit in the cache, then $(K+1)^{th}$ iteration can reuse a cache line from the 1^{st} iteration.

Cache lines for a reference: $RefCost_{intra}(Ref, L_d)$ denotes the total number of cache lines required by a reference Ref in one iteration of L_d, and $RefCost(Ref, L_d)$ denotes the total cache lines fetched by Ref over entire loop L_d. The intra-loop cost of a reference is 1, for the innermost loop. For any intermediate loop, it is the cost (total cache lines fetched) of executing all the iterations of the immediately inner loop.

$$RefCost_{intra}(Ref, L_d) = \begin{cases} 1, & \text{if } L_d \text{ is innermost loop} \\ RefCost(Ref, L_{d+1}), & \text{Otherwise} \end{cases} \tag{2.1}$$

Now for estimating the cost of Ref for the entire loop, there are three cases. If Ref is loop invariant with respect to L, then the cost is same as the cost of a single iteration as estimated above. The second case is when consecutive iterations of L_d refer to same cache line. Let j be the loop index for L_d, $step_j$ as the step increment for L_d, and f_1 be the innermost subscript for Ref. We define a stride as,
$stride(Ref, L_d) = f_1(j + step_j) - f_1(j)$, now the consecutive iterations of L_d refer to the same cache line if the stride is less than cache line size, denoted by the boolean property:

$Consecutive(Ref, L_d) = (stride(Ref, L_d) < CLS)$.

If the static analysis cannot infer any cache line reuse, then each iteration of L_d might fetch a unique cache line. These three cases can be represented as follows,

$$RefCost(Ref, L_d) = \begin{cases} 1 * RefCost_{intra}(Ref, L_d), & \text{if } LoopInvariant(Ref, L_d) \\ \frac{trip}{CLS/stride(Ref, L_d)} * RefCost_{intra}(Ref, L_d), & \text{if } Consecutive(Ref, L_d) \\ trip * RefCost_{intra}(Ref, L_d), & \text{otherwise} \end{cases} \tag{2.2}$$

Loop footprint: $LoopFootprint(L_d)$ denotes the total cache lines fetched for all iterations of L_d. First, for each reference, $Ref \in L_d$, we estimate $RefGroup(Ref, L_d)$. Then we consider a representative reference from each group and denote it as $\mathcal{R}$, the set of all references which fetch a unique cache line, $\mathcal{R} = \{Ref_1, ..., Ref_m\}$. Now, the loop footprint can be estimated as,

$$LoopFootprint(L_d) = \sum_{k=1}^{m} (RefCost(Ref_k, L_d)) \tag{2.3}$$

2.4.3 Set Associative Caches

For modeling the set-associative cache, we use a similar approach as in [29]. Consider a K-way set associative cache of size N bytes with a cache line size of B bytes. So the total number of cache lines can be given by N/B, and total number of sets $S = \frac{N/B}{K}$.

The most common approach to determine the specific cache set that a memory address maps to, is using the least significant bits of the memory address. So given an address $addr$, the cache set can be estimated using a modulo operator, as $(addr/B)\%S$. We are going to exploit the symmetric properties of the modulo operator to simplify our analysis by making the following two assumptions,

1. The array base address is mapped to cache set 0.

2. The array dimensions are known statically

Note that, user options can be provided to select if unique base addresses are mapped to the same cache set or different cache sets and unknown array dimensions are initialized to a user-provided constant.

Consider the code example in Listing 2.3 to understand the implications of the above assumptions.

```
1  A[dim2][dim1][dim0];
2   for (i1=0; i1 < N1 ; i1++)
3    for (i2=0; i2 < N2 ; i2++)
4      for (i3=0; i3 < N3 ; i3++)
5          load A[i1][i2][i3]
6          load A[i1][i2 + 1][i3]
7          load A[i1 + 1][i2][i3]
```

Listing 2.3: Code to illustrate set-associative cache

Assume a set associativity of 4, total 8 sets and a cache line size of 32 elements. Figure 2.4 shows, for $\langle i_1 = 0, i_2 = 0, i_3 = 0\rangle$ there are 3 cache references. If address of A[0][0][0] is $Base$, then its cache line is $\frac{Base}{32}$. The cache line for A[0][1][0] is $\frac{Base+dim_0*1}{32}$ and for A[1][0][0] is $\frac{Base+dim_1*dim_0}{32}$. Now the corresponding cache sets can be estimated as, $Set(Base) = \frac{Base}{32}\%8$, $Set(Base+dim_0*1) = \frac{Base+dim_0}{32}\%8$, $Set(Base+dim_0*dim_1) = \frac{Base+dim_1*dim_0}{32}\%8$.

All of which map to the same cache set, if the array dimensions are divisible by 8.

Now, given that 32 elements fit in the same cache line, the next cache line is fetched only in iteration $\langle i_1 = 0, i_2 = 0, i_3 = 32\rangle$.

Conflict Misses Next, for estimating the conflict misses, the analysis should determine when a cache set is full and eviction can occur.

Consider the Listing 2.4, where all the 5 accesses in the loop body are mapped to the

	Set 0	Set 1	Set 2	Set 3	Set 4	Set 5	Set 6	Set 7
Line 0	A[0][0][0]	A[0][0][32]	A[0][0][64]					
Line 1	A[0][1][0]	A[0][1][32]	A[0][1][64]					
Line 2	A[1][0][0]	A[1][0][32]	A[1][0][64]					
Line 3								

Figure 2.4: Illustration of a 4 way set associative cache

same cache set. Since, there are only 4 slots in the cache set, the fifth access will evict the earliest cache line corresponding to $A[i_1][i_2][i_3]$ as per the LRU policy. As a result, in the next iteration of loop i_3, $A[i_1][i_2][i_3+1]$ is no longer a hit in the cache even though it refers to the same cache line.

```
1  A[dim2][dim1][dim0];
2   for (i1=0; i1 < N1 ; i1++)
3    for (i2=0; i2 < N2 ; i2++)
4      for (i3=0; i3 < N3 ; i3++)
5          load A[i1][i2][i3]
6          load A[i1][i2 + 1][i3]
7          load A[i1 + 1][i2][i3]
8          load A[i1][i2 + 2][i3]
9          load A[i1 + 1][i2 + 1][i3]
```

Listing 2.4: Code to illustrate conflict misses

We use Algorithm 1 to determine the set of references that incur a conflict miss. Given a set of references, each of which access a unique cache line the algorithm first determines the corresponding cache set using the function $getCacheSet$. The array $CacheSetEntries$ records the total number of references mapped to the each cache set. If the total references mapped to any cache set exceeds the set-associativity, then all references mapped to that set are assumed to incur conflict miss. The algorithm does not consider the sequence of memory references, since later passes can potentially reschedule them.

Algorithm 1: $estimateConflictMisses$

```
Data: S, SetAssoc, References
//S: total number of cache sets
//SetAssoc: Set associativity
//References: set of references accessing unique cache lines
Result: ConflictMiss
//ConflictMiss: Set of references that incur conflict miss
//Total cache lines mapped to each set is initialized to 0
1 Let, CacheSetEntries[S] = 0 ∀S ;
2 foreach Ref ∈ References do
3     Let, Set =getCacheSet(Ref) ;
4     CacheSetEntries[Set] + +;
  //For each cache set, if the total lines mapped is greater
    than associativity then it is considered to incur conflict
    miss
5 foreach Ref ∈ References do
6     Let, Set =getCacheSet(Ref) ;
7     if CacheSetEntries[Set] > SetAssoc then
8         Let, ConflictMiss = ConflictMiss ∪ Ref ;
9 return ConflictMiss
```

2.4.4 Memory Cost after Unroll-Jam

In this section we estimate the total number of cache lines required by a loop nest after the unroll-jam transformation, without actually performing the transformation.

By default, without any analysis, the unroll-jam transformation can create one instance of the instruction for every unroll-jam factor of each loop. That is, given an unroll jam factor of $\{u_1, ..., u_n\}$, the total instances of a reference Ref created after the transformation can be expressed as,

$$CopiesReq(Ref, \{u_1, ..., u_n\}) = u_1 * u_2 * ... * u_n \tag{2.4}$$

We will use the term "copies required" to refer to instances of an instruction created within the loop body after the unroll-jam transformation.

Table 2.1: Reuse exposed by unroll-jam

Scalar Reuse	Cache Line reuse

```
1  //Before Unroll-Jam
2  for (i)
3   for (j)
4    x = A[i][j]
5    y = A[i-1][j]
```

```
1  //Before Unroll-Jam
2  for (i)
3   for (j)
4     x = A[j][i]
```

```
1  //After Unroll-Jam (2,1)
2  for (i)
3   for (j)
4    x1 = A[i][j]
5    y1 = A[i-1][j]
6    x2 = A[i+1][j]
7    y2 = A[i][j]
8    // y2 Can Reuse x1
```

```
1  //After Unroll-Jam ((2,1)
2  for (i)
3   for (j)
4    x1 = A[j][i]
5    x2 = A[j][i+1]
6    // x2 can reuse the
7    // cache line for x1
```

Reuse after Unroll-Jam Next consider the case when unrolling exposes a reuse opportunity. This reuse can either be a scalar reuse or cache line reuse as illustrated in Table 2.1.

Our analysis relies on dependence analysis to detect such reuse opportunities statically. Depending on the unroll-jam factor and the dependence distance between two memory references, the total memory references within the loop body after the transformation can be fewer than Equation 2.4. We will derive the equation for possible reuse using the following example 3 level loop nest of Listing 2.5.. There is a reuse opportunity after unroll-jam if

```
1  for (i1=0; i1 < N1 ; i1++)
2   for (i2=0; i2 < N2 ; i2++)
3     for (i3=0; i3 < N3 ; i3++) {
4        load A[f(i1,i2,...,in)] \\ Load1
5        load A[g(i1,i2,...,in)] \\ Load2
6        ...                     \\ Other instructions
7     }
```

Listing 2.5: Sample Code fragment

there is any input dependence between the two array load instructions in Listing 2.5. Let there be a dependence distance of (d_1, d_2, d_3), between the two loads, which can also be expressed as,

$g(i_1+d_1, i_2+d_2, i_3+d_3) = f(i_1, i_2, i_3)$. Listing 2.6 illustrates the result of unroll-jam loop transformation of the Listing 2.5 by a factor of (d_1+2, d_2+2, d_3+2).

```
1  for (i1=0; i1 < N1 ; i1+= d1 + 2)
2   for (i2=0; i2 < N2 ; i2+= d2 + 2)
3    for (i3=0; i3 < N3 ; i3+= d3 + 2) {
4    {// Copy 0
5       load A[f(i1, i2, i3)]
6       load A[g(i1, i2, i3)]
7    }
8    ...
9    {// Copy d1 * d2 * d3
10      load A[f(i1 + d1, i2 + d2, i3 + d3)]
11      load A[g(i1 + d1, i2 + d2, i3 + d3)]
12   }
13   {// Copy (d1 + 1) * (d2 + 1) * ... * (dn + 1)
14      load A[f(i1 + d1, i2 + d2, i3 + d3)]
15      load A[g(i1 + d1 + 1, i2 + d2 + 1, i3 + d3 + 1)]
16   } }
```

Listing 2.6: After unroll-jam of Listing 2.5

Due to the input dependence, the highlighted load instructions access the same array location, and thus the second highlighted instructions can reuse the result of the first load. Here unroll-jam exposes a data reuse opportunity, which can be expressed as, $A[g(i_1 + d_1, i_2 + d_2, i_3 + d_3)] == A[f(i_1, i_2, i_3)]$

Table 2.2: Total copies of a memory references required after unroll jam, estimated using Equation 2.5

Code Snippet	Distance	Total Copies
1 `for (i)` 2 `for (j)` 3 `for (k)` 4 `A[i][j]`	A[i][j]→A[i][j] $\{d_i = 0, d_j = 0, d_k = 1\}$	CopiesReq(A[i][j])= $U_i * U_j * d_k + U_i * d_j * U_k + d_i * U_j * U_k - d_i * d_j * d_k = U_i * U_j$
1 `for (i)` 2 `for (j)` 3 `A[i][j] +` 4 `A[i+1][j+2]`	A[i][j]→ A[i+1][j+2] $\{d_i = 1, d_j = 2\}$	CopiesReq(A[i+1][j+2]) $= U_i * U_j$; CopiesReq(A[i][j])= $d_i * U_j + U_i * d_j - d_i * d_j = U_j + 2 * U_i - 1$

$A[g(i_1 + d_1 + 1, i_2 + d_2 + 1, i_3 + d_3 + 1)] == A[f(i_1 + 1, i_2 + 1, i_3 + 1)]$. Thus any unroll factor greater than (d_1, d_2, d_3), will not require more copies of the "Load2" instruction than already shown in Listing 2.6.

Memory Instruction Copies Required We derived a formula to consider the reuse opportunities exposed after unrolling and estimate the total copies of a memory reference required after the transformation.

Given an n dimensional memory reference Ref and the corresponding dependence vector $(d_1, d_2, ..., d_n)$, Equation 2.5 can be used to estimate the total copies of the reference after unrolling by $(U_1, U_2, ..., U_n)$.

If $U_x \geq d_x + 1 \forall x$, then

$$CopiesReq(Ref, \{U_1, ..., U_n\}) =$$
$$d1 * U2 * ... * U_n + U1 * d2 * ... * U_n + ... + U1 * U2 * ... * d_n$$
$$- d_1 * d_2 * ...d_n \quad (2.5)$$

Table 2.2 shows two simple example uses of this equation.

2.4.5 Register Pressure and Spills

In this section, we estimate the extra memory accesses required due to register spills. We present an algorithm to estimate the register pressure after the unroll-jam transformation by a given factor without actually performing the transformation.

Estimating Register Pressure Given any sequence of instructions, all the live variables require a register. We count the maximum number of variables that are live at any point in the program to estimate the maximum number of registers required for executing that sequence of instructions. We use the standard liveness dataflow analysis to estimate the maximum set of live variables at any point in every basic block.

Register spills after unroll-jam Algorithm 2 is used to estimate the total register spills after the unroll-jam transformation. It takes as input the maximum set of variables that were live at any point of a basic block and estimates the total copies of each of the variables required after unroll-jam by a given factor. We repeatedly call the algorithm 2 for the max-live-set of all the basic blocks in the function. If the register pressure of a type of variable is greater than the available registers in the hardware, it results in spills. The algorithm estimates the total spills for each kind of register.

2.5 Evaluation

2.5.1 Experimental Setup

We used the following three machines as our evaluation platform,

1. IBM POWER9, 3800.0MHz, Linux 4.14.0-115.21.2.el7a.
   ppc64le, 314 GB RAM

2. AMD Ryzen 9 3900X, 2202.539MHz, Linux 4.18.0-041800-generic, 31 GB RAM

Algorithm 2: Total spills after unroll-jam

Data: $MaxLiveVars$, $(U_1, U_2, ..., U_n)$
Result: $TotalSpills$
//Maximum set of live variables: $MaxLiveVars$
//Unroll-jam factor: $(U_1, U_2, ..., U_n)$
foreach $Var \in MaxLive[NextBB]$ **do**
    $TypeOfVar = getTypeOf(Var)$;
    //For each loop level
    foreach $Depth \in \{1, 2, ..., n\}$ **do**
        if Var *isLoopIndependent(*$Depth$*)* **then**
            $VarCopies[TypeOfVar] * = 1$;
        else
            $VarCopies[TypeOfVar] * = U_k$;
    $VarCopies[TypeOfVar] = Min(MaxInterleaveFactor, VarCopies)$;
    $RegisterPressure[TypeOfVar] + = VarCopies$;
//For all different types of registers(Float/Vector/...)
foreach $Type \in RegisterPressure$ **do**
    $HWregisters = getAvailableRegistersOfType(Type)$;
    if $HWregisters < RegisterPressure[Type]$ **then**
        $TotalSpills[Type] = RegisterPressure[Type] - HWregisters$;
return $TotalSpills$;

3. Intel(R) Xeon(R) Gold 6226R CPU, 1200.009MHz, Linux 4.15.0-112-generic, 376 GB RAM

All three machines had an 8-way set associative L1 Data cache of 32KB size. The cache line sizes were 64B for the Intel and AMD machines, and 128B for the IBM machine.

We evaluated OptiMemReuse on all 30 benchmarks in the PolyBench suite [27]. These benchmarks ran in the range of 10ms-100ms for their largest standard inputs (EXTRALARGE). To obtain larger execution times on our modern processors, we increased the input sizes for all benchmarks so that they instead ran in the range of 10s-100s. We used the same input sizes on all the three platforms. We implemented OptiMemReuse as a standalone pass in LLVM, clang version 12.0.0. This pass iterates over all the loop nests in a function, and computes optimized unroll factors for each loop nest, after which the existing LLVM unroll-jam implementation is used to perform the transformation with these factors.

We compiled the baseline version of all benchmarks using "clang -O3 -mllvm -enable-

unroll-and-jam -mllvm -allow-unroll-and-jam", which enables the default unroll-jam transformation. The baseline version uses the existing LLVM implementation for estimating the optimal unroll-jam factors and applying the transformation. So, the only difference in optimizations in our comparison is between the unroll factors selected by the baseline LLVM implementation and those chosen by our OptiMemReuse analysis pass. We also conducted experiments to study the effect of unroll-jam on SIMD vectorization. Other than the default run, we ran experiments with two more configurations, vectorization disabled: "-fno-slp-vectorize -fno-vectorize" and only SLP vectorization enabled: "-fno-vectorize" .

We report performance improvement as the runtime ratio of $\frac{Baseline}{OptiMemReuse}$ (Higher the better). Each benchmark's runtime was measured as the mean of 5 runs with a maximal accepted variance of 5%.

Figures 2.5, 2.6 and 2.7 summarize the performance improvement obtained on IBM POWER9, Intel Xeon and AMD Ryzen, respectively. The geometric means shown are for all 30 benchmarks. Due to space limitations, the charts only show individual bars for benchmarks that resulted in performance improvements greater than 10%, which are in general different for different platforms. Further, the worst-case performance degradation across all 90 data points was 2% (performance improvement of 0.98×).

For Power9, we observe that there is almost no difference in performance with the vectorization enabled. This could be a limitation in the LLVM vectorizer for the Power9 architecture. But on AMD and Intel platforms, for some benchmarks like the "mvt" the unroll-jam transformation exhibits significant opportunities for SLP vectorization. This is because unroll-jam of outer loops exposes more instructions in the loop body, that can be used as seed instruction for SLP vectorization. The performance improvements obtained by the use of OptiMemReuse were observed to be up to 2.57× for IBM POWER9 (geometric mean of 1.21×), up to 4.62× for AMD Ryzen (geometric mean of 1.33×), and up to 5.26× for Intel Xeon (geometric mean of 1.16×).

Since our analysis is parametric in terms of different architecture features like the avail-

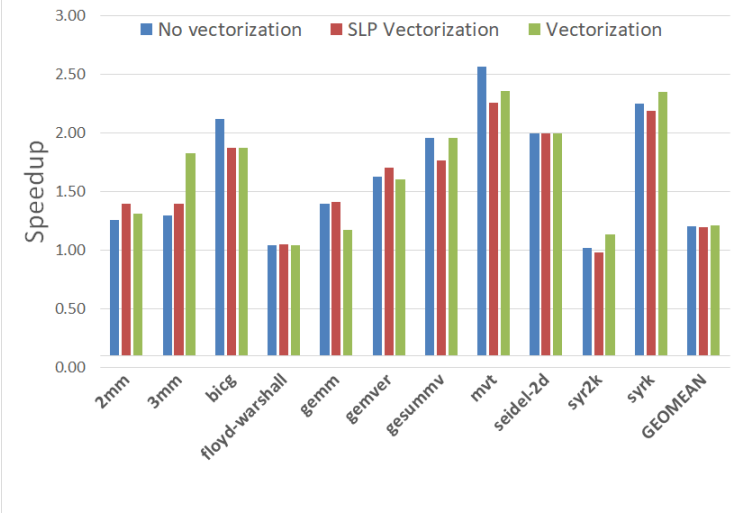


Figure 2.5: Performance Improvement for POWER9

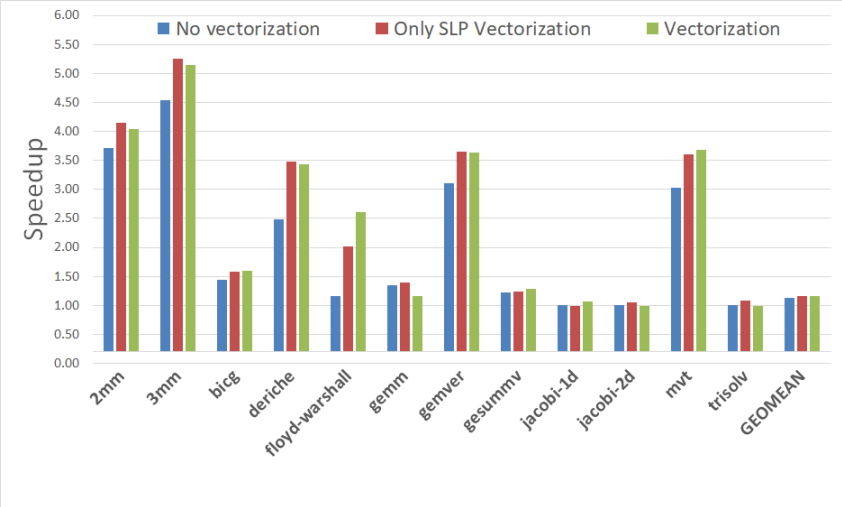


Figure 2.6: Performance Improvement for Intel Xeon

able registers and cache line size, it is able to estimate different unroll-jam factor for the different platforms.

Table 2.3 shows the correlation of various hardware performance counters measured using PAPI and the performance improvement obtained by the unroll-jam transformation for the AMD Ryzen platform. For each of the benchmarks we measured the performance counters for the baseline and OptiMemReuse cases, and the table shows the ratio $\frac{Baseline}{OptiMemReuse}$ for few selected performance counters. This table thus shows the factor by which each of the performance counters decreased for OptiMemReuse compared to the baseline. It

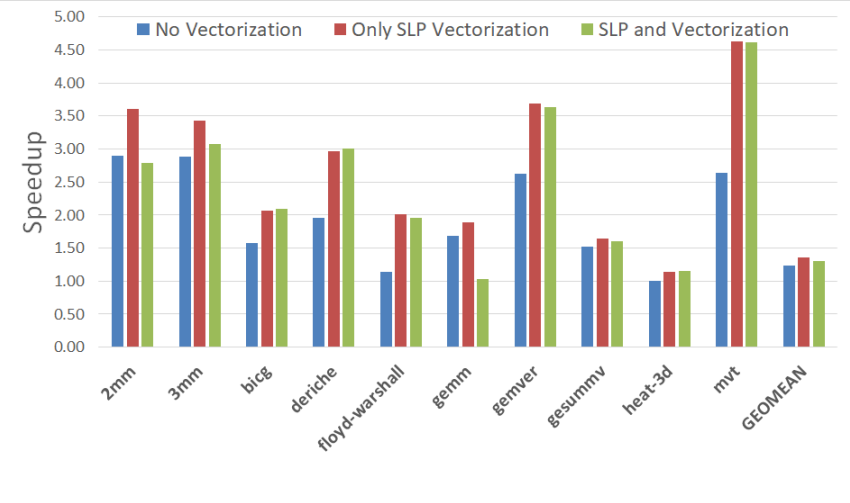


Figure 2.7: Performance Improvement for AMD Ryzen

Table 2.3: $\frac{Baseline}{OptiMemReuse}$ ratio of hardware performance monitors measured on AMD Ryzen platform.

Apps	Total Cycles	L2 Data Cache Reads	Stall Cycles	Branch Taken	Total Instructions
2mm	2.88	6.29	3.11	30.28	1.32
3mm	3.13	10.61	3.40	43.05	1.21
bicg	2.10	1.99	2.13	16.00	1.71
deriche	3.27	3.82	3.40	4.00	1.77
floyd-warshall	2.96	3.93	1.44	64.02	1.72
gemm	1.10	1.65	1.17	8.00	0.93
gemver	3.62	3.58	3.72	16.00	1.68
gesummv	1.64	1.32	1.51	16.00	1.63
heat-3d	1.13	1.01	1.09	1.78	1.31
jacobi-2d	1.05	1.00	1.04	1.33	0.98
mvt	4.57	4.62	4.79	10.67	1.76
trisolv	0.99	1.00	0.99	1.00	1.00

Table 2.4: % Slowdown in compilation time by OptiMemReuse compared to baseline clang (for apps where the slowdown was $> 10\%$)

Apps	Slowdown %
cholesky	33.35
deriche	20.48
doitgen	14.49
heat-3d	39.91
lu	55.49
ludcmp	25.73
seidel-2d	57.92

clearly demonstrates that reduction in L2 data cache reads due to unroll-jam by OptiMemReuse always translates into a performance improvement in runtime. We can also observe that unroll-jam leads to a reduction in total branches taken as expected and in many cases also reduction in total instructions issued. Finally, Table 2.4 shows the percentage slowdown in compilation time (also measured on the AMD Ryzen platform) due to the use of OptiMemReuse. We measured the compilation time as the total elapsed time of the make command, and only show results for apps where the slowdown was ¿10% . As we can see, the largest slowdown was 58%. In most cases, it is not the time spent in the OptiMemReuse pass that contributes to the slowdown. Instead, the increase in code size due to unroll-jam can impact the compilation time of all successive passes.

2.6 Related Work

Cache Modeling There has been a lot of work on cache modeling and estimating the cache misses of a given nest of loops. Ferrante et al.[3] and Sarkar [4] introduced a model to approximate the number of distinct accesses and distinct lines accessed by multiple array references in a loop nest. More recently Gysi et al.[5] presented a symbolic counting technique to accurately estimate the cache misses of an application. Most of such prior work has ignored the set-associativity and conflict misses, which was an important motivation for our work. Additionally they do not consider register spills as part of the memory cost model.

Agarwal et al. [6] and Harper et al. [37] developed analytical models for set-associative caches. Abstract interpretation has also been used to model set associative LRU caches by Martin et al.[7] . Ghosh et al.[8] have derived cache miss equations for perfect loop nests with data dependencies represented by reuse vectors. Chatterjee et al.[9] presented an approach to use Presburger formulas to count cache misses for set associative caches. And most recently Bao et al.[29] presented the PolyCache, an analytical model for set-associative caches for polyhedral programs with static control flow.

Most of the prior work mentioned above have focused on the accurately estimating the cache misses of an application. While in this work, our objective was to develop a simple and approximate model to estimate the relative change in cache misses due to the different unroll-jam factors and finally guide the loop transformation.

Loop transformations for improving data locality Similarly there has been considerable amount of work done in using cost models to guide loop transformations. Wolf et al.[10] addressed the problem of the choice of loop transformations by developing a cost model to increase data locality. Kennedy et al.[11] proposed a simple memory model which optimizes for cache line reuse and selects a loop ordering for generating parallel code for shared-memory multiprocessors.

Carr et al.[28] proposed a very simple cost model to estimate cache misses incurred by a loop nest and use it to guide several loop transformations like permutation, fusion, distribution and reversal. However their work only considered reuse across the innermost loop and also does not account for conflict misses in set-associative caches. Our work is most closely related to their algorithm which is also implemented in the LLVM infrastructure.
Bondhugola et al.[12] have developed a cost model driven automatic transformation framework, PLuTo. They use polyhedral model for optimizing regular programs and find affine transformations for efficient tiling to optimize for parallelism and locality. Shirako et al.[38] employed DL model [4] for optimal tile size selection problem. Qasem et al.[39]

presented an auto-tuning approach by using a detailed analytical model that characterizes the interaction of loop fusion and tiling, across a memory hierarchy for different architectures.

Loop Unroll Sarkar[31] presented a cost model to select unroll factors for perfectly nested loops. The cost model considers the total load store instructions and ILP exposed by the unroll-jam transformation. It also considers register spills and instruction cache capacity constraints to restrict the replace space. Unlike their approach, in OptiMemReuse the cost model tries to minimize the L1 data cache misses and can even select unroll factors that increase the total number of register spills.

Leather et al.[40] presented a profile driven approach for selecting best unroll-jam factor. They developed a sequential sampling plan to automatically adapt to the experiments so that the best optimization settings (e.g., loop unrolling factor) can be determined only with required number of profiling runs for the optimization. Baskaran et al.[41] proposed a compilation framework for automatic parallelization and performance optimization of affine loop nests on GPGPUs with various loop transformations including multi-level loop tiling and unrolling, where the loop unrolling factor is selected by a model-driven empirical replace. Stephenson et al.[42] presented a machine learning approach to predict the optimal unroll factor. They use multi-class classification and support vector machines, to guide the compiler decisions. Domagała et. al[43] demonstrate an approach of combining loop unrolling and instruction scheduling to reduce register pressure of a loop. Murthy et al.[44] develop an approach to identify optimal unroll factors for GPGPU programs. Barua et al. [24] presented a cost model for selecting unroll factors for guiding the thread coarsening transformation for GPU kernels. Most recently Rocha et al.[45] presented a loop unrolling heuristic that optimizes for opportunities of SLP vectorization. There has been a significant amount of work on loop unrolling, each trying to optimize a different metric. Unlike the prior work, our contribution is in the precise modeling of set-associative data cache for dif-

ferent unroll factors and considering register spills into account to infer the best unroll-jam configuration.

2.7 Summary

Past approaches to cache models for compilers have been used to drive code transformations such as loop permutation, loop fusion, loop distribution, and loop reversal to improve memory hierarchy usage. At the same time, unrolling of multiple nested loops (also referred to as "unroll and jam" or "unroll-jam") has emerged as an important code transformation to improve register locality and instruction-level parallelism. As memory cost continues to increase in relative significance in modern CPU architectures, there is a growing need to design robust cost models that can be used to drive unroll-jam transformations by simultaneously taking cache misses and register pressure into account.

In this chapter, we introduce OptiMemReuse, a new static analysis to guide the selection of unroll-jam transformations with the objective of reducing the memory cost of a loop nest while also taking register pressure into account. To do so, we had to extend past work on cache models to also consider outer (non-innermost) loops, as well as constraints such as limited set associativity that are important when modeling real hardware. OptiMemReuse can be used to estimate the cache misses that would occur after performing the unroll-jam transformation for a given unroll configuration, without actually performing the transformation.

We evaluated OptiMemReuse on three hardware platforms – IBM POWER9, AMD Ryzen 9, and Intel(R) Xeon(R) Gold – by implementing it in LLVM and comparing the effect of unroll-jam driven by the OptiMemReuse cost model with the baseline model currently used in LLVM. The results obtained across all 30 PolyBench benchmarks are very encouraging. There was no degradation of more than 2% across all the platforms. The performance improvements obtained by the use of OptiMemReuse were observed to be up to 2.57× for IBM POWER9 (geometric mean of 1.21×), up to 4.62× for AMD Ryzen

(geometric mean of 1.33×), and up to 5.26× for Intel Xeon (geometric mean of 1.16×).

These results suggest that OptiMemReuse can be incorporated in any compiler that performs unroll-jam, so as to deliver significant performance improvements as a result.

CHAPTER 3
STATIC GPU MEMORY BANDWIDTH MODELING TO IMPROVE BANDWIDTH UTILIZATION VIA THREAD COARSENING TRANSFORMATION

3.1 Introduction

Experienced programmers can achieve significant performance and energy efficiency from Graphics processing units (GPUs). They require device-specific program tuning to exploit sufficient memory & computation parallelism. A key challenge in GPU computing is the improvement of programmability: reducing programmers' burden in writing/tuning low-level GPU programming languages such as CUDA [46] and OpenCL [47] without sacrificing performance. OpenACC [48] and OpenMP [49] are programming standards designed to address this challenge. They provide simple directive-based programming with a higher level of abstraction and performance portability across heterogeneous CPU/GPU systems. [50]

In contrast to the low-level GPU programming approach, a directive-based approach like OpenACC relies on the compilers for the design space exploration and optimizations for the target architecture. Application developers can apply the OpenACC directives on legacy C/Fortran applications that are optimized for CPUs. Thus compilers need to consider the architectural differences between CPUs and GPUs and apply program transformations accordingly. A notable difference is in the paradigm for instruction pipelining, CPU out-of-order vs. GPU in-order executions.

The motivation behind the GPU SIMT programming model is to launch a large number of threads, and to rely on the throughput processing model to hide all intra-thread latencies. However, through our experiments, we observed that creating large numbers of

threads can lead to sub-optimal GPU performance in many cases. This is in part because only a limited number of threads can be available for context switching due to hardware resource constraints and they may not suffice to provide the desired latency hiding, thereby leading to processor stalls as threads wait for memory accesses and other long-latency operations to complete. A large number of threads are often queued, waiting for some of the already launched threads to finish. So, if most of the allocated/launched threads are waiting on a long-latency operation, then the hardware has to stall. It cannot switch to the queued threads until currently executing threads finish and release their resources. Thus even though the programmer might expect a large number of threads to keep the hardware busy, only a fraction of them might be running. This results in inefficient usage of the GPU resources if the individual threads do not have enough parallelism to exploit the available GPU throughput.

With this motivation, we propose a cost-driven thread coarsening transformations that exploit sufficient per-thread instruction level parallelism (ILP) for optimal latency hiding. Given an OpenACC program with user-specified loop parallelism, our approach partially converts the loop parallelism into ILP available within a thread/warp by loop unrolling and interleaving transformations. Our proposed cost model predicts the optimal unrolling factor to maximize GPU memory throughput for a given kernel. The main contributions of our work are as follows:

- We propose the use of automatic compiler-generated unroll-and-interleave (referred to as "interleave" in this chapter) loop transformations to perform thread coarsening.
- We introduce a new performance modeling cost function to statically estimate the available memory level parallelism in a GPU kernel, and show how a compiler can use this cost model to automatically select the best interleave factor for a given input kernel.
- We validate our approach on a range of GPU benchmarks available as OpenACC

programs, and show that the interleave factors selected by our cost model are close to optimal in practice. We also show that GPU memory bandwidth utilization is low in many of the benchmark kernels, thereby providing a strong motivation for our cost-driven thread coarsening approach.

The rest of the chapter is organized as follows. Section 3.2 motivates the problem addressed in this chapter. Section 3.3 provides an overview of our approach including the problem statement. Section 3.4 presents the proposed cost model that determines the optimal interleave factor of given kernel. Section 3.5 provides our implementation details in the OpenARC compiler. Section 3.6 presents experimental results to evaluate our approach on the two GPU systems. Section 3.7 and 3.8 summarize related work and our conclusions.

3.2 Motivating Examples

We use three kernels, an array-copy kernel taken from the Rodinia cfd benchmark, a simple matrix multiplication kernel, and an FFT kernel from the NAS parallel benchmarks, to motivate our approach.

3.2.1 Cfd kernel

```
1 #pragma acc kernels loop gang worker independent
2  for (i_c=0; i_c<nelr*5;i_c ++ ) {
3   old_variables[i_c]= variables[i_c];
4  }
```

Listing 3.1: Baseline For cfd Kernel

```
1 #pragma acc kernels loop gang worker independent
2  for (i_c=0; i_c<nelr*5; i_c +=2 ) {
3    old_variables[i_c]=variables[i_c];
4    old_variables[i_c+1]= variables[i_c+1];
5  }
```

Listing 3.2: cfd kernel after unrolling by a factor of 2

We ran the benchmarks on an NVIDIA Tesla P100 processor and measured various performance metrics using nvprof. We also note that this GPU supports a peak memory bandwidth of around 500GBps.

The input sequential C code for Rodinia cfd, annotated with an OpenACC pragma is shown in Listing 3.1. In this code snippet, every iteration of the loop maps to a single thread. Thus every thread(warp) has one memory read followed by a write. That is, every warp stalls after issuing a single memory read transaction, until the memory load is complete.

Now let's consider the achieved average throughput metrics for this kernel as shown in Table 3.1. Firstly we can see that the DRAM read/write throughput almost matches the requested throughput for the kernel. Secondly, we note that this simple kernel is not even able to utilize 25% of the available 500GBps memory bandwidth.

Table 3.1: Throughput Stats for cfd baseline

DRAM Read Throughput	Global Load Requested Throughput	DRAM Write Throughput	Global Store Requested Throughput
113 Gbps	113 Gbps	103 Gbps	113Gbps

Table 3.2: Throughput Stats for cfd after Unrolling

	Speedup	Increase in DRAM Read Throughput by factor	Increase in DRAM Write Throughput by factor
Unroll Factor 2	1.8	1.49	1.43
Unroll Factor 4	2.3	1.49	1.39
Unroll Factor 8	2.3	1.52	1.40

As noted earlier, the GPU SIMT model tries to hide stall latency by creating a sufficiently large number of active warps on an SM. However, since the maximum number of active warps is limited by several hardware resource constraints on an SM, how can we hide the stall latency more effectively?

Our approach is to exploit the observation from the experiments, viz., the GPU memory system can sustain much higher memory bandwidth than what our simple kernel can achieve.

To address this problem, we unroll the openacc annotated 'for-loop' in the code Listing 3.1. The loop is unrolled by a factor of 2, as shown in Listing 3.2.

Now, since the second statement is independent of the first, every warp can issue 2 memory transactions before switching to the next warp. In this case, loop unrolling can effectively double the achieved memory level parallelism.

The performance metrics after unrolling by a factor of $2, 4$ and 8 are shown in Table 3.2. The DRAM throughput utilization increased by 1.5 times, and that in-turn provides a speedup of more than 2 times.

This simple kernel motivates the effectiveness of the loop unroll transformation by increasing the ILP within each warp. Unrolling increases the number of independent instructions available to the warp scheduler within an SM, even if the maximum active warps are limited by other constraints.

3.2.2 Matrix multiplication kernel

Our second example, Listing 3.3 is a basic matrix multiplication kernel, and Listing 3.4 shows the kernel after unrolling the outermost loop.

```
1  {
2   #pragma acc  kernels loop  gang worker independent
3   for (i=0; i<2048; i ++ ) {
4    for (j=0; j<2048; j ++ ) {
5     float sum;
6     sum=0.0F;
7     #pragma acc  loop seq
8     for (k=0; k<2048; k ++ ) {
9      sum+=(b[((i*2048)+k)]*c[((k*2048)+j)]);
10    }
11    a[((i*2048)+j)]=sum;
12   }
13  }
14 }
```

Listing 3.3: Baseline for Matrix Multiplication Kernel

We repeated the last experiment with this kernel, and Table 3.3 shows the ratio of baseline to the unrolled kernel for few measured stats. Note that it contradicts with our cfd kernel observations since the runtime deteriorated and there is no difference in bandwidth utilization.

```
1  #pragma acc  kernels loop  gang worker independent
2  for (i=0; i<2048; i=(i+2)) {
3   for (j=0; j<2048; j ++ )     {
4    float sum;
5    sum=0.0F;
6    #pragma acc  loop seq
7    for (k=0; k<2048; k ++ ) {
8     sum+=(b[((i*2048)+k)]*c[((k*2048)+j)]);
9    }
10   a[((i*2048)+j)]=sum;
11   {
12    float sum;
13    sum=0.0F;
14    #pragma acc  loop seq
15    for (k=0; k<2048; k ++ ) {
16     sum+=(b[(((i+1)*2048)+k)]*c[((k*2048)+j)]);
17    }
18    a[(((i+1)*2048)+j)]=sum;
19   }
20  }
21 }
```

Listing 3.4: Unroll Factor=2, Matrix Multiplication

Unlike the Listing 3.2, it is not obvious that Listing 3.4 has increased the ILP after unroll. We should note that the GPU micro-architecture is an in-order issue pipeline. Hence the architecture relies on sophisticated and expensive compiler instruction scheduling to increase the ILP, which might not always be feasible. We have observed that for a majority of the

Table 3.3: Ratio of nvprof stats, Baseline to Unroll ($\frac{baseline}{unroll}$)

Runtime Ratio	DRAM Read Throughput Ratio	Read Transactions Ratio	DRAM Write Throughput Ratio	Write Transactions Ratio
0.992	0.999	0.998	1.000	1.00091

Table 3.4: nvprof Stats for matrix multiplication after Interleaving

	Speedup	DRAM Write Throughput Ratio, $\frac{interleave}{baseline}$	DRAM Read transactions Ratio, $\frac{interleave}{baseline}$
Unroll Factor 2	1.6	1.4	0.5
Unroll Factor 4	2.2	1.3	0.2
Unroll Factor 8	2.6	1.4	0.1

benchmarks unrolling does not result in a speedup.

Next, we apply the loop interleaving transformation, as shown in Listing 3.5. This transformation re-orders statements after unrolling, such that the next iteration's statement is interleaved with the current iteration. The Table 3.4 shows the effect of interleaving on the runtime and the corresponding nvprof stats that explain the speedup. We get up-to 2.6 times of speedup after interleaving by a factor of 8. Listing 3.5 shows that the array index for memory read c is the same for the interleaved instructions. That means after interleaving the number of memory reads to c are halved. The total number of memory reads decreases by almost 80% with the interleaving factor of 8 (Table 3.4). This is a very well known property of matrix multiplication that has been exploited by various optimizations like tiling and using shared-memory. In this chapter, we propose a simpler transformation and relatively cheaper analysis to achieve significant speedup.

Our third example is a Fourier transformation benchmark from FT, NAS parallel benchmark. For this benchmark, interleaving by a factor of 2 provides a 10% improvement in runtime, but further interleave factors result in slowdown. The Table 3.5 shows the throughput and speedup for this kernel. And we can see, that when interleaving by a factor larger than 2, the memory bandwidth becomes the bottleneck, and results in a slowdown.

The three example benchmarks show that the memory access pattern and the dependen-

Table 3.5: nvprof Stats for ***fft*** after Interleaving (higher ratio is better performance)

	Speedup ($\frac{baseline}{interleave}$)	DRAM Read Throughput Ratio, ($\frac{interleave}{baseline}$)	DRAM Write Throughput Ratio, ($\frac{interleave}{baseline}$)
Unroll Factor 2	1.10	1.16	1.13
Unroll Factor 4	0.71	0.80	0.78
Unroll Factor 8	0.38	0.45	0.45

cies within a kernel influence the performance significantly.

```
1  #pragma acc  kernels loop  gang worker independent
2  for (i=0; i<2048; i=(i+2)) {
3   for (j=0; j<2048; j ++ ) {
4    float sum;
5    float sum0;
6    sum=0.0F;
7    sum0=0.0F;
8    #pragma acc  loop seq
9    for (k=0; k<2048; k ++ ) {
10     sum+=(b[((i*2048)+k)]*c[((k*2048)+j)]);
11     sum0+=(b[(((i+1)*2048)+k)]*c[((k*2048)+j)]);
12    }
13    a[((i*2048)+j)]=sum;
14    a[(((i+1)*2048)+j)]=sum0;
15   }
16 }
```

Listing 3.5: Interleave Factor=2, Matrix Multiplication

3.3 Overview

3.3.1 Problem Statement and Cost Model

We saw in section 3.2 that the memory access pattern and memory bandwidth utilization of a kernel have a significant impact on its performance. In this chapter, our objective is to analyze a given loop kernel and apply the loop interleave transformation to improve its memory transaction efficiency. We formulate the problem statement as follows: *Given an input loop kernel, select an interleaving factor that maximizes GPU resource utilization, thereby delivering the best speedup from the loop interleave transformation.*

The first step is given a specific GPU, determine the number of concurrent memory accesses required to saturate the memory bandwidth (lets denote it by $concurrency_{required}$). Next, given a parallel loop annotated with an OpenACC pragma, estimate the maximum number of concurrent memory accesses issued by the corresponding kernel. Let us denote this by $concurrency_{achieved}$. Also, $concurrency_{interleaved}(factor)$ denotes the estimated concurrency for a given interleaving $factor$. Now our objective is to maximize the GPU memory bandwidth utilization.

$$ratio = \frac{concurrency_{estimated}(factor)}{concurrency_{required}} \tag{3.1}$$

We quantify the bandwidth utilization as $B\_utilization$ in Equation 3.2.

$$B\_utilization(factor) = \begin{cases} 0, & \text{if } factor = 1 \\ ratio, & \text{if } ratio < 1 \\ -1, & \text{otherwise} \end{cases} \tag{3.2}$$

Now, whichever factor gives the maximum $B\_utilization$, we can choose that interleave

factor (Equation 3.3).

$$best\_factor = \underset{factor \in \mathbb{Z}+}{\arg\max}(B\_utilization(factor)) \tag{3.3}$$

The first case of Equation 3.2 makes sure that an interleave factor of 1 (do not interleave) is selected,

if $\forall factor \geq 2, ratio > 1$. That is the baseline version already has enough concurrency to saturate the GPU memory bandwidth.

3.3.2 Loop Interleave Transformation

In this section, we provide details of our proposed loop interleave transformation. Consider the example code in Listing 3.6. It has two statements within the loop body. The i used in $S1(i)$ refers to the fact that this statement belongs to loop iteration i. Now let's perform the loop unroll transformation, to obtain the Code in Listing 3.7.

```
1  for(i=0;i<N;i++){
2    S1(i)
3    S2(i)
4  }
```

Listing 3.6: Example for-loop

```
5  for(i=0;i<N;i+=2) {
6    S1(i)
7    S2(i)
8    S1(i+1)
9    S2(i+1)
10 }
```

Listing 3.7: Unrolled for-loop

Now, if we interleave the statements after unrolling, we get the code in Listing 3.8. This loop interleave transformation that was proposed in [51], is legal only if there is no loop carried dependency between the statements, $S2(i)$ and $S1(i+1)$. In general, if a loop has no loop-carried dependencies, then that is sufficient to ensure that interleaving is legal. We plan to use the loop interleaving transformation to maximize bandwidth utilization within

each thread in a GPU kernel. Let us assume that the user annotated the above for loops as independent loop kernels. As has been noted in [51], in general operations within a single iteration are dependent on each other, and exhibit reduced parallelism. The unroll transformation can potentially increase the ILP within an iteration, but the compiler needs dependence analysis to determine that $S1(i+1)$ is independent of $S1(i)$. This information is already available to us when we unroll a parallel loop since $S1(i+1)$ and $S1(i)$ belong to different iterations. This problem of not exploiting available ILP within each thread/iteration is amplified on a GPU since it has an in-order issue pipeline. The interleaving transformation, therefore, unlocks the potential increase in ILP after unrolling. The loop interleave transformation is a generalization of the thread coarsening transformation proposed by Unkule et. al [52]. Even though loop interleaving has not been studied in the context of GPUs, thread coarsening is a very well researched topic. There exist several machine learning models to predict the optimal thread coarsening factor [53, 54, 55].

```
11   for(i=0;i<N;i+=2)
12   {
13   S1(i)
14   S1(i+1)
15   S2(i)
16   S2(i+1)
17 }
```

Listing 3.8: Interleaved increment by 2

```
18 for(i=0;i<N/2;i++)
19 {
20   S1(i)
21   S1(i+n/2)
22   S2(i)
23   S2(i+n/2)
24 }
```

Listing 3.9: Interleaved halved bound

Preserving Coalesced Accesses

In addition to re-ordering statements within an iteration after unrolling, we can also re-order across iterations since we are dealing with a parallel for loop. For example, the code Listing 3.9, shows one such option. This variation of the interleave transformation is

similar to the stride option for thread coarsening proposed by Magni et al[54].

Interleaving can, in general, destroy the coalesced access patterns. For example an access $m[i]$ in Listing 3.6 would be a coalesced access. But after interleaving (Listing 3.8) the access is no longer fully coalesced, since both the $m[i]$ and $m[i+1]$ accesses would occur inside the same iteration/thread. Interleaving can thus potentially double the number of memory transactions in Listing 3.8.

To preserve the coalesced access, consecutive iterations must have a stride of 1. Listing 3.9 shows such a transformation, in which the loop upper bound is halved and the cloned statement refers to $i + n/2$ instead of $i + 1$.

3.3.3 Modeling the GPU Architecture

This section is based on the Chapter 3 and 4 of Volkov's Thesis [17].

Required Concurrency

In this section we see an example of how to estimate the number of concurrent instructions required to saturate the Arithmetic and Memory Bandwidth.

Firstly we can use Little's law [56] to equate throughput, latency and concurrency as shown in Equation 3.4.

$$Throughput = \frac{Concurrent\ Instructions}{Instruction\ Latency} \tag{3.4}$$

For each SM, peak instructions issued per cycle(IPC) is:

$$Max\ Arithmetic\ Throughput = \frac{\#FP\ Units}{\#Warp\ Size} \tag{3.5}$$

Using same numbers from [17] for Maxwell architecture (128 FP units, 32 instructions per warp, and 6 cycle FP latency), we can estimate the number of concurrent single precision floating point instructions required to achieve the peak IPC as $\frac{128}{32} * 6 = 24$.

We can do a similar estimate for the memory instructions. Consider a memory pin bandwidth of $224\ GB/s$ and a clock rate of $1.266\ GHz$ i.e., the DRAM can transfer $\frac{224}{1.266} \approx 177\ Bytes/cycle$. Now effective bandwidth for 16 SMs is $\frac{177}{16} \approx 11\ Bytes/cycle$. To convert $Bytes/Cycle$ into IPC, we need to consider the Total Bytes transferred for each instruction. If a warp issues a coalesced single precision floating point memory transaction, then it results in a single $128\ Byte$ transaction from DRAM. So, the peak Memory Instruction throughput is $\frac{11}{128} \approx 0.086\ IPC$. The DRAM access latency for this $128\ Byte$ access is about $368\ cycles$ on Maxwell. Finally, the number of memory instructions required to achieve this throughput is $(0.086\ IPC) * (368\ cycles) \approx 32\ instructions$.

Modeling the Latency, Throughput and Arithmetic Intensity

This section estimates the concurrency achieved by a kernel, as opposed to the required concurrency to saturate the available bandwidth. Assumptions about GPU micro-architecture,

- The GPU limits the maximum number of warps that can be launched by the kernel.
- The resource requirements like Shared Memory/Registers per warp limit the maximum number of warps running in parallel.
- Whenever a warp is blocked due to some data dependency, it switches to another ready warp.
- Only the warps that are running in parallel are available for context switching.

We assume a kernel, with a mix of single precision floating point instructions and single precision memory instructions. The code has a pattern whereby each thread reads certain memory values, performs computations on them, and finally stores the result to memory. We model the worst case scenario whereby all the instructions miss the L1/L2 cache and every memory instruction accesses the DRAM.

Let mem_lat be the memory access latency for a fully coalesced DRAM access from a warp, single precision floating point instruction latency be alu_lat and α be the arithmetic intensity. Consider the following dependency graph of an instruction sequence that is executed by every warp.

$$Mem \rightarrow ALU_1 \rightarrow ALU_2 \rightarrow \cdots \rightarrow ALU_\alpha \rightarrow Mem \ldots$$

Each warp can have a repeating pattern of such instruction sequences. After a warp finishes execution the next warp is launched. Thus we analyze the case, where the above instruction sequence is either repeated by the same warp or different warps.

Equation 3.6 denotes the latency for the $\alpha + 1$ sequence of instructions.

$$latency_{\alpha+1} \geq (mem\_lat + \alpha * alu\_lat) \tag{3.6}$$

Now, a memory instruction is executed every $latency_{\alpha+1}$ cycles, and if n warps are being executed in parallel on the SM, then the sustained memory instruction throughput can be expressed as Equation 3.7.

$$mem\_throughput \leq \frac{n}{latency_{\alpha+1}} \tag{3.7}$$

Given that α floating point instructions are executed for every memory instruction, we get Equation 3.8.

$$arith\_throughput = \alpha * mem\_throughput \tag{3.8}$$

Let the constants, max_ALU_thru and max_MEM_thru be the memory and arithmetic throughput bounds for a given GPU. We can express this as

$$mem\_throughput \leq MIN(n/latency_{\alpha+1}, max\_MEM\_thru, \frac{max\_ALU\_thru}{\alpha}) \tag{3.9}$$

This shows that, when the number of concurrent memory instructions is low, then the program is latency bound, i.e., low throughput utilization. Also, if α is too big, then the application is ALU throughput bound, that is the memory latency is essentially completely hidden by the ALU instructions.

Given the ALU instruction mix and the GPU throughput bounds,the number of concurrent warps required to hide the latency and maximize the throughput can be expressed as

$$n \geq latency_{\alpha+1} * MIN(max\_MEM\_thru, \frac{max\_ALU\_thru}{\alpha}) \tag{3.10}$$

The above equation reveals that we need more concurrent warps if α is low, that is a memory bound kernel. While a compute bound kernel (high α) will require fewer concurrent warps to hide the memory latency[17].
Inspired by the above model, we propose a compiler analysis pass, to select the latency bound kernels and apply the loop interleave transformation that can boost its performance by doubling the number of memory accesses issued in parallel and enabling to hide the memory latency.

3.4 Modeling the User Program

3.4.1 Program Dependence Graph

We use a simplified Program Dependence Graph (PDG) [57] to estimate the instruction level parallelism within a thread, i.e., an iteration of a parallel loop selected for GPU execution.. Each node in this PDG is either a memory operation or a floating-point arithmetic operation. A directed edge represents a Read after Write (flow) dependence from its source node to its sink node. For simplicity, we ignore control dependences in our cost analysis, which effectively assumes that all memory and arithmetic operations are performed

Table 3.6: PDG example 1

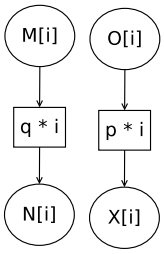
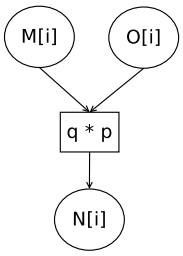
PDG	Instructions
	1 `for (int i=0;i<N;i++) {` 2 `  float q = M[i] ;` 3 `  float p = O[i];` 4 `  N[i] = q * i ;` 5 `  X[i] = p * i;` 6 `}`

Table 3.7: PDG example 2

PDG	Instructions
	1 `for (int i=0;i<N;i++) {` 2 `  float q = M[i] ;` 3 `  float p = O[i];` 4 `  N[i] = q * p ;` 5 `}`

unconditionally.

Definition 3.4.1.1. *Program Dependence Graph* $G : \{V, E\}$ *is a directed graph* G *with nodes* V *and directed edges* E. $(\forall u \in V)$, u *represents either a memory operation or a single /double precision floating point operation.* $(\forall \{u, v\} \in E)$, v *is expected to read a datum written by* u*, as determined by conservative static analysis..*

A PDG may contain disjoint subgraphs, which can easily be identified as its connected components. For example, Table 3.6 and 3.7 show two simple examples of for loops intended for GPU parallel execution, and their corresponding PDGs.

Note that integer operations are not included in the PDG. So, a memory read that only depends on integer arithmetic for index computation will have no incoming edges in the PDG.

Each node in the PDG can have different attributes and associated values as follows:

- "*Node.type* ", defined for all nodes, can have the values: $Memory$, $SinglePrecision$, $DoublePrecision$
- "*Node.Stride* " attribute is the access stride for memory nodes (3.4.2.1).
- The value of the "*Node.transactions* " attribute is the estimated number of 32-byte memory transactions (3.4.2.2).

Definition 3.4.1.2. *Level_Zero_Nodes: is a set of memory nodes that do not depend on any other nodes. It can be defined for a PDG* $\{V, E\}$ *as*

$$u \in Level\_Zero\_Nodes \iff u \in V \;\&\&\; u.Type = Memory \&\& \nexists v : \{v, u\} \in E \tag{3.11}$$

The observation being made here is that each thread reads from one or more memory locations, does some computations on them, then writes to a memory location. Since each thread is working in parallel, the general pattern is multiple independent memory loads, few arithmetic computations, finally a single memory store.

Each independent component of the PDG has multiple level 0 nodes but only single final store node. Since in general, the computed value is stored at a single location, and not multiple locations.

If the number of store nodes is more than the number of level 0 nodes, then our heuristic will underestimate the number of memory transactions issued in parallel. Our model focuses on estimating the cost of the most common computation patterns and optimizes for them.

3.4.2 Memory Transactions

Whenever a warp executes a memory access instruction, it generates a certain number of transactions. Depending on the GPU and the index access pattern, the device memory (global memory) can be accessed via 32-,64- or 128-byte memory transactions [46].

An important consideration for our analysis is to estimate the number of memory transactions incurred by a memory reference instruction.

Let $warp\_size$ be the number of threads in a warp. For NVIDIA GPUs, this number is 32. In the worst case, the maximum number of transactions that can be generated by an instruction is $warp\_size$, if each thread in a warp accesses a different cache line. More generally, we use the following formulation to estimate the number of transactions corresponding to each memory access instruction.

Definition 3.4.2.1. *Access_Stride: Let i be the innermost loop index(corresponding to threadIdx.x in CUDA), and **M[**$f(i)$**]** be a memory reference. Then,*

$$Access\_Stride = \begin{cases} f(i+1) - f(i), & \textit{if it is constant} \\ warp\_size, & \textit{otherwise} \end{cases} \tag{3.12}$$

Definition 3.4.2.2. *Number of Transactions: Let $cache\_line$ be the cache line size and $bytes$ be the size of datum being accessed by the memory reference, e.g., 4 bytes for single precision and 8 bytes for double precision floating-point.*

$$\#transactions = \min(warp\_size, \frac{warp\_size * bytes * Access\_Stride}{cache\_line}) \tag{3.13}$$

In Equation 3.12 if $Access\_Stride$ is not a constant, then in the worst case each thread in a warp can generate a separate transaction, the second case takes care of that. Similarly in Equation 3.13, maximum possible transaction is $warp\_size$.

3.4.3 Estimated Concurrency

Given the PDG for a loop annotated as a kernel, our objective is to estimate the total number of memory transactions issued in parallel by a single SM.

For our purposes, this heuristic is required to decide whether the kernel has enough

memory transactions to hide memory latency efficiently. Hence instead of trying to compute the exact number, we devise a heuristic for a reasonable estimate. After experiments over several benchmarks, we observed that the set $Level\_Zero\_Nodes$ is a reasonable candidate that accounts for most of the concurrent memory transactions. Then the achieved concurrency can be estimated as the total memory transactions issued by the $Level\_Zero\_Nodes$ multiplied by the number of warps.

Let's denote the total number of warps active on an SM by $\#warps\_perSM$. Then,

$$concurrency_{achieved} = \sum_{\forall u \in Level\_Zero\_Nodes} u.transactions * \#warps\_perSM \tag{3.14}$$

Now if we interleave the kernel by $factor$, then the number of transactions can also increase by $factor$, and assuming $\#warps$ remains same, then the concurrency achieved also increases by $factor$. But we also need to consider the loop independent memory accesses. If a memory accesses has a stride of zero, that means two consecutive iterations access the same memory location. So interleaving by a factor of 2, would not double the transactions. We need to subtract the loop independent transactions to estimate the concurrency after interleaving. The access stride (3.4.2.1) property of a memory reference indicates whether it is loop independent or not. Hence we only subtract the references that have a stride of zero.

$$loop\_independent\_transactions = \sum_{\forall u | u.stride == 0} u.transactions * \#warps\_perSM \tag{3.15}$$

Equation 3.16 shows the estimated concurrency after interleaving by $Factor$.

$$concurrency_{estimated}(Factor) = Factor * concurrency_{achieved} - (Factor - 1) * loop\_independent\_transactions \tag{3.16}$$

Finally we use Equation 3.2 and Equation 3.3 from subsection 3.3.1 to estimate the optimal interleave factor.

Arithmetic Intensity

As we saw in subsubsection 3.3.3, Equation 3.10, a kernel can benefit from interleaving only if it is latency bound. We use the arithmetic intensity of a kernel to determine if it is ALU throughput bound. The arithmetic intensity is calculated as the ratio of ALU operations to memory operations. The Equation 3.17 counts the number of single precision operations and two times the number of double precision operations, for total ALU operations.

$$\begin{aligned} ALU\_ops = &|\{\forall u \mid u.Type == SinglePrecision\}| \\ &+ 2 * |\{\forall u \mid u.Type == DoublePrecision\}| \end{aligned} \tag{3.17}$$

$$Mem\_ops = |\{\forall u \mid u.Type == Memory\}| \tag{3.18}$$

$$Arithmetic\_Intensity = \frac{ALU\_ops}{Mem\_ops} \tag{3.19}$$

We use the Equation 3.19 as a filter, to not interleave kernels, that have an arithmetic intensity above a certain threshold. This threshold depends on the GPU and can be calculated as $\frac{max\ ALU\ throughput}{max\ Memory\ throughput}$. But in practice it is lower than this ratio and is determined experimentally.

3.5 Implementation

3.5.1 OpenARC

We used the OpenARC [58] compiler for our implementation. It provides a powerful and flexible infrastructure to experiment with various loop transformations at the source level. OpenARC is a source to source compiler framework built on top of Cetus [59] and supports all features in OpenACC specification $v1.0$. OpenARC provides an Abstract Syntax Tree

Algorithm 1 Interleave a For Loop

Input Annotated For Loop
Output Interleaved Loop

```
1:  function INTERLEAVE(Forloop)
2:      pdg ← Build_PDG(Forloop)
3:      pdg ← Compute_transactions(Forloop, pdg)
4:      factor ← Compute_Factor(pdg)
5:      if factor == 1 then return
6:      for stmt ∈ Forloop.statements() do
7:          if No_Need_to_Copy(stmt) then
8:              continue
9:          toCopy ← factor
10:         while toCopy > 0 do
11:             cloned_st ←clone(stmt)
12:             Fix_Variables(cloned_st, loop_index, toCopy)
                    ▷ Replace and Keep track of new variables,
                        ▷ along with corresponding factor
13:             Insert_Statement_After(cloned_st, stmt)
14:             toCopy ← toCopy − 1
15:         Finalize_Variables(stmt)
```

Figure 3.1: Algorithm to interleave a for loop

Algorithm 2 Compute Interleave Factor for a Loop

Input PDG
Output Interleave factor

```
1:  function COMPUTE_FACTOR(pdg)
2:      gpu_max ← Get_Machine_Model()
3:      factor ← 1
4:      if pdg.arithmetic_intensity > gpu_max.ALU_bound then
    return factor
5:      concurrency ← Estimate_Concurrency(pdg)
                                          ▷ Equation 14
6:      while concurrency < gpu_max.Mem_bound do
7:          factor ← factor + 1
8:          concurrency ← Estimate_Concurrency(pdg,factor)
                                          ▷ Using Equation 16
        return (factor − 1)        ▷ Max factor whose ratio is < 1
```

Figure 3.2: Algorithm to compute interleave factor

intermediate representation, in which the program statements are annotated with user directives. OpenARC has many built-in compiler analyses, and user directive controlled loop transformations such as loop unrolling and loop collapsing. We experimented with the loop unroll transformation, but in general, blindly unrolling (without interleaving) results in a slowdown on average.

OpenARC provides an interesting framework to explore a huge design space of generating a GPU kernel from a parallel for loop. Our heuristic based interleave transform is one such optimization.

3.5.2 Loop Interleave Transform Pass

We implemented the Loop interleave transformation, as a new pass, analogous to the existing unroll transformation in OpenARC. The loop interleave pass is called before any of the other transformations of OpenARC. For example, the loop normalization pass, which fixes the loop index initialization to zero and increment to one, must occur after our pass.
Following standard OpenACC programming conventions, the user annotates a for loop with "#pragma acc kernels loop gang worker independent" to identify that the loop can be parallelized across Grids(Gang) and Threadblocks(Worker). We iterate over all such annotated loops, and apply the interleave transformation selectively, according to our cost model.

Figure 3.1 shows a high-level overview of our loop transformation pass. We make a single pass over the IR to construct the program dependence graph and build data structures for level zero nodes (3.4.1.2).
Given a for loop, we first check whether it is legal to perform the interleave transformation. Even though a loop is marked as loop kernel by the user, there can be specific other annotations and OpenARC loop properties that prevent us from interleaving. For example, if we interleave a for loop annotated with a "reduction" variable, then that would result in two reduction variables after interleaving. OpenARc cannot handle more than one reduction variable currently. Hence we cannot perform interleaving in this case. The statement

2 of the Figure 3.1 handles this case. Then we build the PDG from the loop body. After building the PDG we annotate each node in the graph with its corresponding stride estimate (3.4.2.1) and transactions estimate (3.4.2.2).

Estimating the Transactions

For each memory access, the function $Compute\_Transactions$ estimates the number of transactions issued by using the Equation 3.13. *Note that* $Compute\_Transactions$ *does not modify the AST, this is a symbolic transformation for the purpose of analysis only*.

```
26  #pragma acc kernels loop independent
27  for(i=0;i<N;i++){
28    int x = i+32;
29    #pragma acc loop seq
30    for (j = 0 ; j < M ; j++) {
31      int y =  1024*j
32      int z = j+y+x;
33      a[z] = j;
34    }
35  }
```

Listing 3.10: Example, Compute Transactions

Consider the memory access "$a[z]$" in Listing 3.10. First, we perform induction variable substitution by replacing each variable with expressions which are a function of the indices of the enclosing loop. After substitution, each array index expression is a pure function of the loop indices. So, "$a[z]$" is transformed to "$a[j+1024*j+i+32]$". Now the innermost loop annotated with openACC loop independent pragma (loop i) should correspond to the thread index.

So, the stride as per 3.4.2.1 is,

$$(j + 1024 * j + (i + 1) + 32) - (j + 1024 * j + i + 32) = 1$$

Similarly using 3.4.2.2, $\#transactions = (32 * 4 * 1/128) = 1$

Interleave Factor

Figure 3.2 is an implementation of Equation 3.3, that computes the optimal factor. Firstly we call a function $Get\_Machine\_Model$, that computes the maximum number of ALU ($gpu\_max.ALU$) and Memory ($gpu\_max.Mem$) instructions required to saturate the the GPU memory and ALU bandwidth, as explained in subsubsection 3.3.3. For most practical purposes, the theoretical peak bandwidth is difficult to sustain, hence we lowered the bounds based on observations from several experiments. Then line 4 checks if the arithmetic intensity is already enough to saturate the ALU bandwidth, then return a factor of 1 in that case (i.e. don't interleave). This filters out the compute bound kernels (Equation 3.10). Line 5 initializes $concurrency$ using Equation 3.14 that estimates the number of transactions issued in parallel. The loop on line 6 keep incrementing the interleave factor until the estimated concurrency is enough to saturate the GPU memory bandwidth. Line 8 uses Equation 3.16 to estimate the concurrency after interleaving by $factor$.

Loop Transformation

Once the optimal factor is predicted, the central loop transformation follows from line 6 in Figure 3.1. We iterate over every loop statement and insert "*factor" number of copies of every loop dependent statement. The new clones are added just following the original statement. For the transformation to be legal, we must preserve the fact that every cloned statement, belongs to a unique loop iteration. Line 12, calls a function* fix_variables, which replaces the index variable by the corresponding iteration index to which the cloned state-

ment belongs. For example for factor 2, and loop index i, every occurrence of i is replaced by $i + 1$ in the cloned statement. Line number 7 in Figure 3.1 checks if a statement needs to be copied. If a statement is not directly or indirectly dependent on the loop index, then it need not be copied. These are the loop independent statements.

Loop Independent Conditions

A particular case of loop-independent statements is the loop-independent branch condition. If a branch condition is not dependent on the loop index, or any other iteration private variable, then it is loop-independent. Such a branch condition is not duplicated after interleaving. Only the statements within the branch that are loop dependent are cloned and interleaved. If each loop iteration writes to a loop local variable, then every iteration gets a private copy of the variable. The algorithm substitutes all these loop dependent variables, with the cloned version of that iteration.

Preserving Semantics

One of the most critical and complex functions is $fix\_variable$, because the transformation is being performed on the AST. It has to keep track of every variable declared and subsequently used within the loop body. Whenever a statement is cloned, all the loop-dependent operands of the statement must be fixed. The transformation can change the semantics of the program if we do not preserve the correct dependencies.

As an example Listing 3.5 from section 3.2, shows the effect of applying Figure 3.1. The variable sum is a loop-private variable, and hence two copies of it exist in the interleaved loop. Every original statement refers to loop index i and variable sum. While every cloned statement refers to $i + 1$ and $sum0$, the only two loop dependent variables.

Table 3.8: Benchmarks

Suite	Benchmark	Total Kernels
Rodinia(with and without unified memory) [60]	kmeans, cfd, backprop, hotspot, nw, lud, srad,	33
Scientific	xsbench, randles,	6
NAS parallel (single and double precision) [61]	cg,ep,ft	165
Kernels	matmul, mandlebrot, jacobi, laplace2d	8

Table 3.9: Details of the GPUs

GPU Name	TeslaP100	QuadroM1000M
CUDA version	9.0	9.0
CUDA Capability	6.0	5.0
Global Memory,MBytes	16281	2003
Total SMs	56	4
CUDA Cores/SM	64	128
Total CUDA cores	3584	512
GPU Max Clock rate	1.33 GHz	1.07 GHz
Memory Clock rate	715 Mhz	2505 Mhz
Memory Bus Width:	4096-bit	128-bit

Table 3.10: Summary of Geo-Mean and Max Speedup

GPU Name	TeslaP100	QuadroM1000M
Our Geo-Mean Speedup	1.323	1.202
Oracle Geo-Mean Speedup	1.379	1.26
Our Max Speedup	2.6	5.85
Oracle Max Speedup	2.6	7.76

Table 3.11: Stall Reasons Comparison, and how it changes with interleaving

Stall Reason	Baseline	Interleave 2	Interleave 4	Interleave 8
Execution Dependency	18.886%	18.168%	16.937%	12.730%
Memory Dependency	63.167%	55.987%	51.599%	58.841%
Instruction Fetch	4.954%	4.304%	4.563%	6.732%
Memory Throttle	1.021%	3.112%	5.059%	5.620%

3.6 Evaluation

We used the benchmarks summarized in Table 3.8, supplied with the OpenARC compiler [62], for our evaluation. We ran experiments on two different NVIDIA GPUs, summarized in Table 3.9. Because of the difference in infrastructure with previous works and fair evaluation, we compare our model with an oracle which always predicts the ideal interleave factor (the one that yields maximum speedup).

To build the Oracle model, we ran experiments with Interleave Factors of 1,2,4,8 and 16 and selected the interleave factor that provided the best speedup. Table 3.10 presents a summary of the speedups achieved. One of the significant side effects of the interleaving transformation is reducing the total number of warps, which can help improve efficiency when the available number of CUDA cores is also smaller compared to the number of warps. We observed this with the much higher maximum speedups achieved on the Quadro GPU than Tesla since Quadro has much fewer cores than Tesla. On the other hand, Quadro has a lower Geo-Mean speedup since the maximum memory bandwidth on the GPU is lower compared to Tesla. Hence the opportunity to increase throughput is also lower. Figure 3.3 shows the speedup achieved by our model and the speedup achieved by the oracle model. The arrow below the plot shows the Oracle optimal factor. For most benchmarks our model achieves the same speedup as the oracle model. Table 3.11 compares how the stall reasons change as we increase the interleave factor. Each cell is the average of 215 kernels. For example in the baseline, on average, 18.8% of the stalls were due to Execution Dependency.

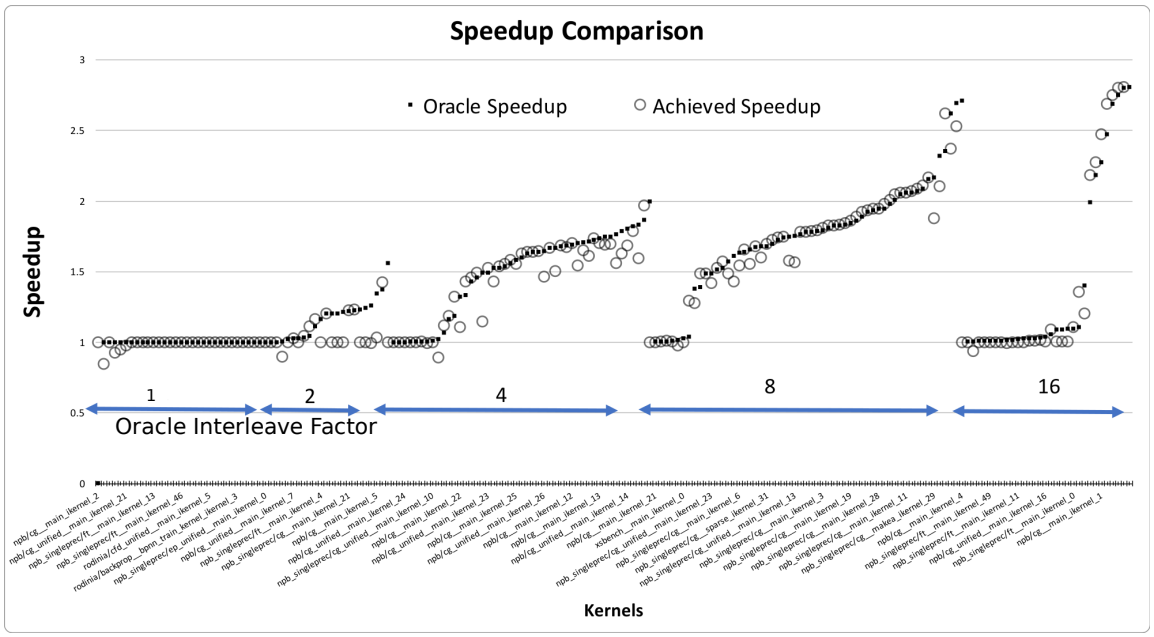


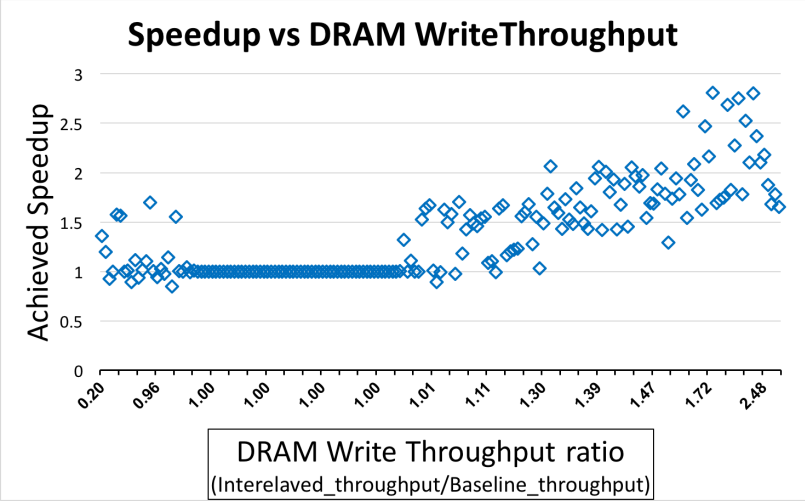
Figure 3.3: P100 Speedup Comparison with Oracle. Blue arrow shows Oracle Factor, and kernels are sorted by Oracle Speedup.

The first row shows that this reduces as we increase the interleave factor. That means fewer instructions were stalled waiting for its input operands to be available, in the interleaved kernels.

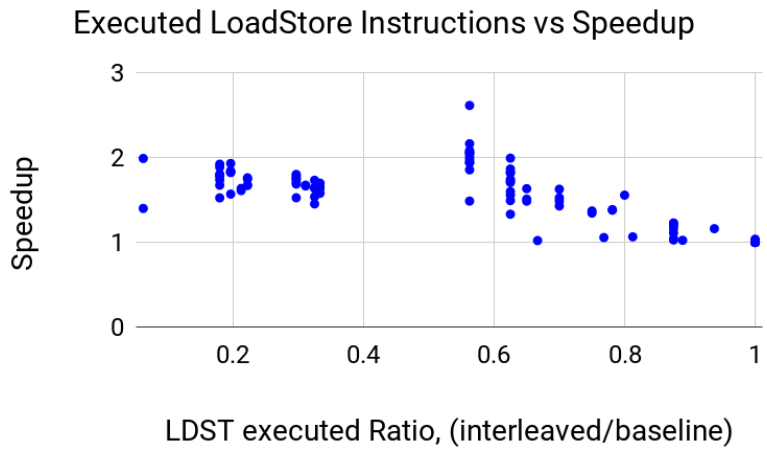
The Memory dependency metric shows that we can hide latency effectively on applying interleaving, since fewer instructions are waiting for a previous memory access to complete as we apply interleaving. The % of stall due to memory dependency decreases on interleaved kernels.

Similarly, stalls due to instruction fetch increase with interleaving, since the number of instructions inside each kernel increased. The next metric, % stall due to outstanding memory requests increases on the interleaved kernels. This also demonstrates the major motivation for this transformation, which was increasing memory bandwidth utilization, is accomplished.

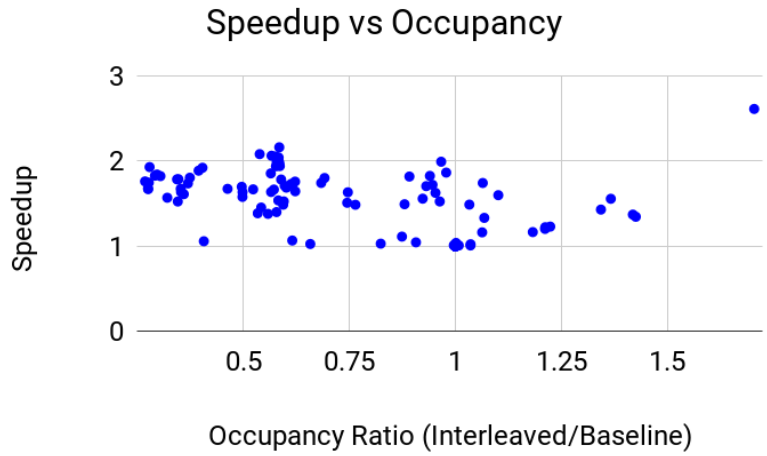
We observed a strong correlation of Load/Store instructions executed and DRAM write throughput with speedup. Figure 3.4a shows how speedup varies with DRAM throughput. The DRAM write throughput increased by almost 4× as we achieved speedup of about



(a) X-axis shows the kernels sorted in increasing DRAM write throughput ratio, rightmost value represents a 4.43 times increase in throughput.



(b) Speedup vs Load/Store Instructions Executed



(c) Speedup vs Occupancy

Figure 3.4: Comparison of Speedup with profiled metrics

2×. Figure 3.4b shows another major reason for the speedup. The total number of load/store instructions greatly reduces after interleaving, if there are enough thread independent memory operations.

One of the major drawbacks of loop unroll/interleaving, in general, is increase in register pressure which reduces occupancy. Even though loop interleave decreases occupancy, since our cost model tries to increase memory level parallelism, the decrease in occupancy still results in speedup. Figure 3.4c shows that we are able to achieve better performance with fewer active threads on each SM, which was again a major motivation for our work.

3.7 Related work

The two main areas of work that are closely related to our work can be found in past work on *cost models* and *thread coarsening* transformations.

In the area of cost models, Volkov et. al [16] discuss the performance analysis of various linear algebra applications on GPUs, that introduced the thread coarsening transformations by hand. Volkov's thesis [17], introduces a simple framework to model GPU performance based on extensive analysis of representative workloads. We use some of the ideas from his thesis to influence our cost model. However, our cost model includes other considerations, such as intra-thread parallelism, that were not considered in his thesis. In addition, our work is focused on using cost models to automatically optimize OpenACC kernels, where as Volkov's work was focused on using cost models to optimize hand-coded GPU kernels and to understand the factors that influence GPU performance.

Hong et. al [63] presented the MWP-CWP model for modeling GPU performance. Subsequently, Sim et al. [64] proposed the GPUPerf framework that uses accurate analytical performance modeling to guide compiler optimizations. They collect data by profiling the CUDA executable, running an Ocelot emulator along with binary static analysis, to drive certain compiler optimizations. Our performance modeling adapts these concepts, with other static cost model considerations, for use in an automatic optimizer for OpenACC

kernels.

In the area of thread coarsening, Unkule et al.[52] presented a compiler framework to perform automatic thread coarsening transformation on CUDA kernels. Their model uses runtime profiles and machine learning to guide the thread coarsening. Likewise, Magni et al. performed an extensive evaluation of the performance benefits of thread coarsening [54], and later introduced a neural network based technique to select an optimized thread coarsening factor [53]. Cummins et. al [55] also introduced DeepTune, a neural network based approach, that learns the program features that affect the performance of the application through extensive profiling and training, through the use of NLP modeling techniques. Their work focused on predicting CPU/GPU execution for a given kernel, as well as the thread coarsening factor for the kernels to be executed on GPUs. In contrast to these past approaches, our approach performs cost analysis statically and automatically without relying on any runtime profile information or machine learning models. Further, all the past approaches showed average performance improvements of under 1.2× on NVIDIA GPUs, whereas our cost-driven approach showed an average (geometric mean) improvement of 1.32× on NVIDIA GPUs.

3.8 Summary

Directive-based programming models like OpenACC provide a higher level abstraction and low overhead approach of porting existing applications to GPGPUs and other heterogeneous HPC hardware. Such programming models increase the design space exploration possible at the compiler level to exploit specific features of different architectures. We observed that traditional applications designed for latency optimized out-of-order pipelined CPUs do not exploit the throughput optimized in-order pipelined GPU architecture efficiently. In this chapter we develop a model to estimate the memory throughput of a given application. Then we use the loop interleave transformation to improve the memory bandwidth utilization of a given kernel.

We developed a heuristic to estimate the optimal loop interleave factor, and implemented it in the OpenARC compiler for OpenACC . We evaluated our approach on over 216 kernels to achieve a Geo-mean speedup of 1.32×.

Our compiler optimization aims to provide the right balance between performance, portability and productivity.

CHAPTER 4

STATIC MODELING OF HOST-GPU MEMORY MOVEMENT FOR OPTIMIZATION

4.1 Introduction

As high-performance computing enters an era of extreme heterogeneity, there is an increasing proliferation of general and special purpose accelerators as well as a concerted effort by higher level parallel programming models to support heterogeneous computing, e.g., OpenMP, OpenACC, X10, Chapel, Julia. Data movement between host and accelerators is a fundamental operation in heterogeneous computing, and parallel programming models vary in supporting data movement either explicitly or implicitly. Data movement is also a major source of overhead, both in execution time and energy. It thus makes sense that minimizing data movement while maintaining the correctness of a program is one of the most important optimizations that compilers and application developers focus on [65, 66, 67, 68, 69]. In this work, we propose a program analysis framework to enable the compiler to detect and remove redundant memory copies automatically.

We use OpenMP 4.5 [1] as an example parallel programming model to demonstrate our optimization framework. We can offload a region of code to accelerators like GPUs using OpenMP. An application developer can specify several different kinds and combinations of OpenMP directives to extract optimal performance from a specific hardware. But the developer also needs to ensure the correctness and absence of data races while manually optimizing the application. Given the complexity of OpenMP specifications, this is a non-trivial task and requires time-consuming efforts from expert programmers. Tools like OmpSan[26] help developers debug incorrect usage of OpenMP memory mapping directives.

[1] www.openmp.org/wp-content/uploads/openmp-4.5.pdf

Our objective is to investigate how the compiler can optimize the memory management operations, while the user only needs to specify synchronization operations for correctness.

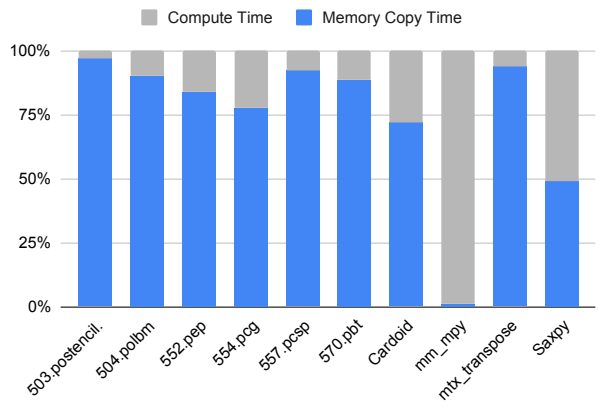


Figure 4.1: Comparison of Compute Time vs Memory Copy time for default memory mapping scheme of OpenMP

Significance of Memory Management. Figure 4.1 shows the significance of the data movement overhead for 10 OpenMP GPU applications. In this experiment, the kernels don't use any explicit memory mapping and rely on the default behavior, which is to copy data from a host to GPU before launching the kernel and back to host after it executes. It compares the % time spent on computing vs. data transfer operations. The experiment illustrates the inefficiency of the default mapping since except for the compute-intensive *mm_mpy* and *saxpy* kernels, over 70% of the time is spent on memory transfer operations alone.

In this work we formalize the data movement optimization problem and define an intermediate representation suitable for analysis of memory accesses and data movements in heterogeneous computing. Then we present our optimization framework that uses the intermediate representation to perform lazy code motion and partial redundancy elimination on data movement operations. The main contributions of this work include:

1. Introduce a general optimization framework to apply partial redundancy elimination, that uses dataflow analysis to identify redundancies in data movement, and a code transformation, lazy code motion, to eliminate the redundancies;

2. Extend the Heap SSA, to Location aware heap SSA (LASSA) to consider heterogeneous memory spaces. Implement the LASSA construction, and the code transformation framework in LLVM tool chain, and evaluate them with real world heteroge-

neous computing applications.

4.2 Background

4.2.1 OpenMP Execution Model

In this section we briefly discuss the OpenMP programming model. We use the term device to refer to a computing resource. The host device is the CPU that begins executing the program. There are optional accelerators like a GPU that are called target devices. An OpenMP program begins as a single thread of sequential execution, called the master thread, which runs on the host device. The OpenMP **target** directive is used to specify a block of code that needs to be offloaded to a device. One or more target devices can be available to the host for offloading code and data. The **target** directive generates a new target task, which may execute on a target device. The target task starts with an initial thread, and teams of threads can be optionally created depending on the usage of other constructs like **teams** and **parallel**.

Memory Space. The most important aspect of the memory model [2] [70] is that the tasks running on the host and the target devices have a separate state that is never shared. Each host device and target devices have at least one attached storage resource(s) that is private to them. This is called a memory space in OpenMP terminology. In case the host and target task need to communicate, they do so by explicitly copying data from one memory space to another. The memory space is a persistent resource, that is, the target memory space retains all the data, unless it is explicitly deleted. OpenMP provides directives to manage each memory space explicitly. The host can allocate, copy, and delete data from a target memory space. OpenMP provides a relaxed-consistency, shared-memory model.

OpenMP can allocate memory from the storage resource of a memory space associated with an allocator. The memory space is persistent.

Every device has an implicit/explicit **target data** region that determines how an original

[2] www.openmp.org/wp-content/uploads/openmp-4.5.pdf

variable in a data environment is mapped to a corresponding variable in a device data environment. The original variable in a data environment and the corresponding variable(s) in one or more device environments may share storage. It can result in data races without intervening synchronization. If a corresponding variable does not exist in the device data environment, then access to the original variable results in unspecified behavior (without unified memory).

Memory model and related issues OpenMP [71], [70] provides a relaxed-consistency, shared-memory model. All OpenMP threads have access to,

1. *memory*, to store and retrieve variables;

2. *temporary view*, which can represent any kind of intervening structure, like registers, cache. It allows threads to cache variables and avoid expensive access to *memory*;

3. *thread private memory*, which is private to each thread, and cannot be accessed by other threads;

There are two kinds of variable accesses: *s*hared and *p*rivate. Each reference to a *s*hared variable becomes reference to the original variable. For *p*rivate variable, a new version of the original variables is created in memory for each task/SIMD lane, corresponding to the directive.

A single access to a variable need not be atomic with respect to the same variable. **Data Race** occurs, if at least one thread reads from a memory unit and at least one thread writes without synchronization to that same memory unit, including cases due to atomicity consideration. If a data race occurs then the result of the program is non-determined.

The OpenMP memory model has relaxed-consistency, because a thread's temporary view of memory is not required to be consistent with memory at all times. To enforce consistency between multiple thread's view of memory, OpenMP introduces the **flush** operation. The completion of a strong flush of a set of variables executed by a thread is

defined as the point at which all writes to those variables performed by the thread before the strong flush are visible in memory to all other threads and the thread's temporary view of all variables involved is discarded.

The completion of a release flush guarantees that any prior operation that writes or reads a shared variable will appear to be completed before any operation that writes or reads the same shared variable that follows an acquire flush with which it synchronizes. Release and acquire flush can be used for thread synchronization. A release flush is the source of the synchronization and an acquire flush is the sink of the synchronization, such that the release flush *synchronizes* with the acquire flush. In this work, we do not consider acquire and release flush. We respect the more stricter, strong flush consistency.

4.2.2 Memory Management

Each host device and target devices, have at least one memory space, which is the attached storage resource(s). OpenMP can allocate memory from the storage resource of a memory space associated with an allocator. The memory space is persistent.

4.2.3 Memory Consistency with Flush Operation

The memory model has relaxed-consistency, because a thread's temporary view of memory is not required to be consistent with memory at all times. OpenMP uses the **flush** operation to enforce consistency between multiple thread's view of memory.

The completion of a strong flush of a set of variables executed by a thread is defined as the point at which all writes to those variables performed by the thread before the strong flush are visible in memory to all other threads and the thread's temporary view of all variables involved is discarded. The completion of a release flush guarantees that any prior operation that writes or reads a shared variable will appear to be completed before any operation that writes or reads the same shared variable that follows an acquire flush with which it synchronizes.

Release and acquire flush can be used for thread synchronization. A release flush is the source of cthe synchronization and an acquire flush is the sink of the synchronization, such that the release flush *synchronizes* with the acquire flush.
In this work, we donot consider acquire and release flush. We respect the more stricter, strong flush consistency.

4.2.4 OpenMP Happens Before relation

An operation op_X simply happens before an operation op_Y if any of the following conditions are satisfied:

1. Sequential Execution: op_X and op_Y are performed by the same in-order executed thread *T*, and op_X precedes op_X in the thread's program order.

2. Synchronization: op_X synchronizes with op_X according to the flush synchronization conditions explained above or according to the base language' memory model definition of synchronizes with, if such a definition exists.

3. Transistancy: there exists the third operation op_Z, such that op_X simply happens before op_Z and op_Z simply happens before op_Y.

4.2.5 OpenMP Memory Consistency

The observable completion order of memory operations, as seen by all threads is guaranteed according to the following rules

- If two operations performed by different threads are sequentially consistent atomic operations or they are strong flushes that flush the same variable, then they must be completed as if in some sequential order, seen by all threads.

- If two operations performed by the same thread are sequentially consistent atomic operations or they access, modify, or, with a strong flush, flush the same variable,

then they must be completed as if in that thread's program order, as seen by all threads.

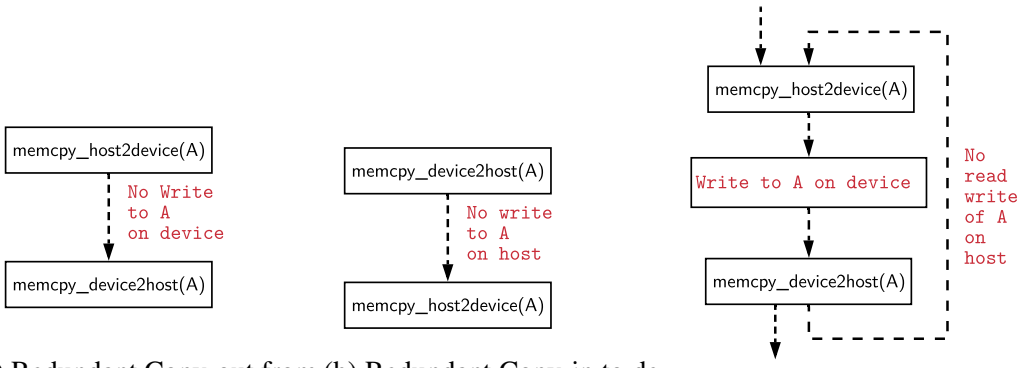
- If two operations are performed by different threads and one happens before the other, then they must be completed as if in that happens before order, as seen by all threads, if :
  - both operations access or modify the same variable,
  - both operations are strong flushes that flush the same variable, or
  - both operations are sequentially consistent atomic operations
- Any two atomic memory operations from different atomic regions must be completed as if in the same order as the strong flushes implied in their respective regions, as seen by all threads.

4.2.6 Heap SSA Form

A heap SSA from [72] is an intermediate representation, that is extended from Array SSA form [73] and models each access of the disjoint memory space as a distinct logical "heap array". Heap SSA employs use $u\phi$ and $d\phi$ operators to chain memory load and store operations respectively. It was originally designed for strongly typed language, but it is also applicable to weakly typed language by introducing an uniform heap array that captures element-level dataflow information for heap data structures, e.g. arrays.

4.3 Motivation

Figure 4.2 shows some typical cases of redundant memory copies that programmers need to detect and optimize manually. Here, $memcpy\_host2device$ copies an array from host to device, while $memcpy\_device2host$ copies it back from the device to host. It shows a dummy CFG in which the dotted line represents an arbitrary sequence of code, which respects the condition mentioned alongside it.



(a) Redundant Copy-out from device

(b) Redundant Copy-in to device

(c) Redundant copies within loop

Figure 4.2: Common patterns of redundancy

Redundancy Pattern 1. Figure 4.2a is the simplest use case; if a kernel launched on the device does not update an array, then there is no need to copy the array back to the host. The default behavior of OpenMP target constructs is to copy in and out every array.

```
1  int A[10];
2  #pragma omp target map(A)
3  {
4    for (i = 0 ; i < 10; i++)
5      A[i] = i;
6  }
7  print(A)
8  #pragma omp target map(A)
9  {
10   for (i = 0 ; i < 10; i++)
11     A[i] + = i;
12 }
13 print(A)
```

Listing 4.1: Default memory map

```
1  int A[10];
2  #pragma omp target data map(tofrom:A)
3  {
4   #pragma omp target map(alloc:A)
5   {
6    for (i = 0 ; i < 10; i++)
7      A[i] = i;
8   }
9   #prargma omp target update from(A)
10  print(A)
11  #pragma omp target map(alloc:A)
12  {
13    for (i = 0 ; i < 10; i++)
14     A[i] + = i;
15  }
16 }
17 print(A)
```

Listing 4.2: Explicitly specify data copies

Redundancy Pattern 2. Figure 4.2b shows the second pattern, when a copy from host to device is redundant since the array is already the latest version on device. After executing a kernel on device, we copy the array back from device to host. Listing 4.1 shows this coding pattern using OpenMP target offloading constructs. Line 2 and line 8 launch a kernel on device that updates the array A in device memory. The print statement on line 7 only reads the array and the array is not updated on host before launching the second kernel. Since the host did not update the array, the device already has the latest version of the array, and the copy is redundant. Listing 4.2 shows the usage of *target data map* clause on line 2 to handle such redundancies. We explicitly leave the array on the device persistent memory for later use. The line 11 kernel launch no longer copies the array back to device.

This example motivates our claim that optimizing even simple memory copy redundancies requires nontrivial understanding of OpenMP spec and the knowledge of all the available directives and their possible usage.

Redundancy Pattern 3.Figure 4.2c shows another pattern when a kernel is launched on the device inside a loop, and we copy the data from the host to device and back to

```
1 int A[10];
2 for (t = 0 ; t < 100; t++) {
3   #pragma omp target map(A)
4   {
5     for (i = 0 ; i < 10; i++)
6       A[i] += i;
7   }
8 }
9 print(A)
```

Listing (4.3) Kernel Launch within loop

```
1  int A[10];
2  #pragma omp target data map(tofrom:A)
3  {
4   for (t = 0 ; t < 100; t++) {
5     #pragma omp target map(alloc:A)
6     {
7       for (i = 0 ; i < 10; i++)
8         A[i] += i;
9     }
10  }
11 }
12 print(A)
```

Listing (4.4) Explicit memory copies

Figure 4.3: Redundant copies within loop, Pattern 3

host redundantly in every iteration. Listing 4.3 shows the OpenMP example for the third case, the target construct on line 3 copies the data from host to device before launching the kernel on device and back to host after the kernel returns. But, since the outer loop of line 2, executing on host does not access the array, both the copies are loop-invariant. That is, we can move the host to device memory copy before the loop, and device to host memory copy after the loop. Listing 4.4 shows the usage of memory map environments to remove the redundancy.

Even though we show very simple examples here, these patterns can be generalized to complex real world use cases. The dotted line of the CFG represents the fact that there

can be arbitrary function calls across several different source files and even the pair of memory copies can be present in different functions. This can make the manual detection of redundant memory copies and its optimization much more complicated and error-prone.

We have observed this issue of the memory copy management over various feedback from OpenMP application developers. The common uses cases are usually scientific applications with large legacy codebases, that are being ported to GPUs using the OpenMP *target offloading* feature launched in version *4.5*. The nontrivial effort required for manual memory management is our motivation to develop a compiler optimization to automate removal of such redundant memory copies.

4.3.1 Challenges

To solve the problem introduced above, we need to address the following challenges:

- Representation of concurrent memory accesses over the same or different arrays;
- Reasoning about the definition-use(def-use) relationship of array accesses across different memory spaces;
- Whole program analysis, that infers the optimal program points for inserting memory copy operations, and detects redundant data movements (Interprocedurally).

4.4 Our Approach

Problem Statement Based on the programming-model, first, the compiler needs to identify where to insert the memory copy operations to ensure correctness. Then an analysis is required to determine partially and fully redundant memory copies. Finally, a code transformation is needed to remove all the redundancies.

Proposed Solution We design an intermediate representation to express the memory model of the programming paradigm and develop an analysis based on that representa-

tion, to optimize the number of memory copies between different memory spaces. We make the following basic assumptions

- We assume the pointer analysis can disambiguate the named arrays. If the alias analysis fails to identify each array uniquely, our optimization fails.
- To keep the analysis simple, any element-level access is conservatively assumed to access the entire array. This constraint can be removed by doing an index range analysis for each array access.

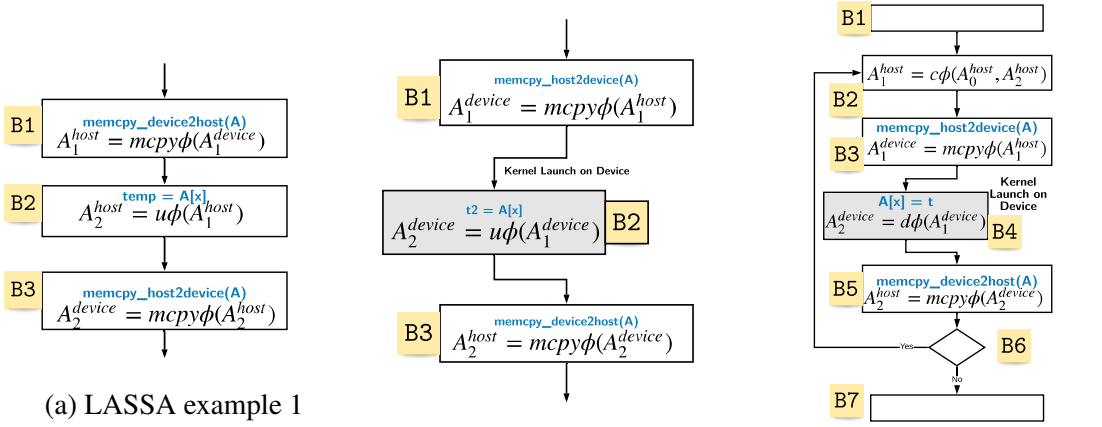
4.4.1 Location Aware Heap SSA

The heterogeneous computing patterns mainly deal with array-based data structures over one or more memory spaces of different devices. In this section, we introduce Location-Aware Heap SSA(LASSA), that extends the Heap SSA to consider the memory space in which each array resides. To uniquely identify each array access in the LASSA, we create a new version of the array for every corresponding access to it. We define LASSA operators that map an array version in one memory space to another array version in the same or different memory space. We call these array versions as a definition.

We use the notation, D_i^r, to denote the i^{th} definition in memory space r.

Definition 1. *We define the following operators in the LASSA:*

1. $D_i^r = d\phi(A, D_j^r)$ *creates a new definition* D_i *of an array* A*, such that,* D_j^r *is the prevailing definition of A just prior to* D_i^r *in same memory space* r*.*
2. $D_k^r = c\phi(A, \{D_i^r, D_j^r\})$*, creates a control merge of the definitions* $\{D_i^r, D_j^r\}$*, for the array* A*;*
3. $D_i^r = u\phi(A, D_j^r)$*, denotes the read of array A*
4. $D_i^r = mcpy\phi(A, D_j^p)$*, creates a new definition of array* A*, due to a copy from memory space* p *to memory space* r*, this is a new operator in extension to heap SSA;*



(a) LASSA example 1

(b) LASSA example 2

(c) LASSA example 3

Figure 4.4: Example LASSA operators, shaded blocks are executed on device

The semantics of the $d\phi$ and $u\phi$ operators are associated with the respective memory write and read operations. The $u\phi$ operator also generates a dummy definition, for array reads. The main purpose of the $u\phi$ operator is to remove redundant copy statements, that have no following use of the array. The control merge operator $c\phi$ merges the reaching definitions from two incoming paths and creates a new definition. The $u\phi$, $d\phi$ and $c\phi$ are the same operators from [72]. A $mcpy\phi$ is associated with a program point where the memory from source memory space data is flushed/written out to the destination memory space. *This guarantees the copied data is visible to any following memory operations.* So, we can use $mcpy\phi$ for both synchronous or asynchronous memory copy. Still, the placement of the operator depends only on when the actual write is visible, as defined by the memory concurrency model. Next we discuss some example LASSA representation.

Case 1 Figure 4.4a shows an example LASSA for case 1. Basic Block $B1$ copies data back from the device to host, assuming there is some preceding kernel that executes on the device not shown here. Assuming A_1^{device} is the most recent version of the array on the device, the copy creates A_1^{host}, a new version of the array on host represented by $A_1^{host} = mcpy\phi(A_1^{device})$. Next, $B2$ reads a location of the array on host, represented by the $u\phi$ operator. Finally, $B3$ uses the $mcpy\phi$ operator to denote the copy from host to device.

Case 2 Figure 4.4b shows the LASSA for case 2. $B2$ is a kernel executed on device, denoted by the shaded block in the figure. $B1$ denotes the host to device memory copy with the $mcpy\phi$ operator, and it updates the version of the array on device to A_2^{device}. After the copy, A_1^{device} is the updated version of the array on device read by the $u\phi$ operator of $B2$. $B3$ copies the array back to host after $B2$ finishes execution on device.

Case 3 In Figure 4.4c $B4$ is a kernel launched on device, which is executed inside a loop. This represents the loop invariant case. $B3$ copies the array from host to device, and $B5$ copies the array back from device to host. $B2$ is the entry block of the loop, it merges the control from the back edge. Assuming A_0^{host} is the last version of array on host before entry to loop, the $A_1^{host} = c\phi(A_0^{host}, A_2^{host})$ merges the A_2^{host} from loop body to create a new version A_1^{host}. $B4$ updates the array on device, denoted by the $d\phi$ operator which creates the version A_2^{device}, that is copied back to host at $B5$.

4.4.2 Redundancy

We will use the data flow analysis defined in section 10.3 of the standard compiler textbook [74] for partial redundancy elimination[75, 76] of memory copy between different memory spaces. In this section, we define the data flow properties in terms of the $mcpy\phi$ LASSA operator.

Definition 2. ***Availability****: An $mcpy\phi$ of A is said to be available between two memory spaces* m *and* p*, at a basic block* B*, if any memory copy of A between* m *and* p *is redundant at* B *since both memory spaces have the same version of the array after the last copy. This is a forward analysis.*

Availability implies, after the last copy: $D_i^m = mcpy\phi(A, D_j^p)$, D_i^m is still the most recent version of the array *A* on memory space *m*, and D_j^p is the most recent version of array *A* on memory space *p*. We use the same definition of $AvailOut$ from [74], it is

Table 4.1: Transfer Functions for the Basic Block Local Properties

LASSA Operators	Downward Exposed	Upward Exposed	Killed Copy
Explanation	if $A_{\{p,q\}} \in DEExpr(B)$ then, the version of A on p and q are same at the end of B.	if $A_{\{p,q\}} \in UEExpr(B)$ then, copy from p to q can be hoisted up at the head of B	Killed Copy
Initialization	$DEExpr(B) = \{\}$	$UEExpr(B) = \{\}$	$ExprKill(B) = \{\}$
Analysis Direction	Forward	Backward	Forward
$D_i^r = d\phi(A, D_i^r)$	$DEExpr(B) \setminus A_{\{r,x\}}\forall x$	$UEExpr(B) \setminus A_{\{r,x\}}\forall x$	$ExprKill(B) \cup A_{\{r,x\}}\forall x$
$D_i^r = u\phi(A, D_j^r)$	$DEExpr(B)$	$UEExpr(B)$	$ExprKill(B)$
$D_i^r = mcpy\phi(A, D_j^q)$	$DEExpr(B) \cup A_{\{q,r\}}$	$UEExpr(B) \cup A_{\{q,r\}}$	$ExprKill(B)$

initialized to $AvailOut(Entry) = \phi$,

$$AvailOut(n) = \bigcap_{m \in preds(n)} (DEExpr(n) \cup (AvailOut(m) \cap \overline{ExprKill(m)})$$

Here, $DEExpr(n)$ is the set of downward exposed $mcpy\phi$ operators, defined in Table 4.1.

Definition 3. ***Anticipability****: An* $mcpy\phi$ *of* A *is anticipable(very busy) between memory spaces* m *and* p*, on exit of a basic block* B*, if every path that leaves* B*, executes a memory copy of* A *between* m *and* p*, and it is legal to hoist it to the end of* B.

Anticipability is a backward analysis, computed using the following equations,

$$AntIn(m) = (UEExpr(m) \cup (AntOut(m) \cap \overline{ExprKill(m)})$$

and $AntOut(exit) = \phi$, $AntOut(n) = \bigcap_{m \in succ(n)} AntIn(m)$

To compute the availability and anticipability, we define a lattice over the $mcpy\phi$ of array variables. We use $A_{\{src,dst\}}$ to denote that the memory copy of A between src and

	Available Out
Figure 4.4a $B1$	$A_{\{host,device\}}$
Figure 4.4a $B2$	$A_{\{host,device\}}$
Figure 4.4b $B1$	$A_{\{host,device\}}$
Figure 4.4b $B2$	$A_{\{host,device\}}$

(a) Redundancy

	Available Out	Anticipable In
$B1$	ϕ	$A_{\{host,device\}}$
$B2$	ϕ	$A_{\{host,device\}}$
$B3$	$A_{\{host,device\}}$	$A_{\{host,device\}}$
$B4$	ϕ	ϕ
$B5$	$A_{\{host,device\}}$	$A_{\{host,device\}}$
$B6$	$A_{\{host,device\}}$	ϕ
$B7$	$A_{\{host,device\}}$	ϕ

(b) Partial Redundancy

Figure 4.5: Computing Availability and Anticipability

dst is redundant, that is both memory spaces have exactly the same copy of A. Then, as per the data flow equations from [75] and [74], we use the local properties in Table 4.1 to compute the availability and anticipability.

Definition 4. ***Redundancy****: A copy statement between memory spaces* m *and* p *for a particular array* A *is redundant, if both the memory spaces already have the same version of* A.

So, a memory copy, $D_i^p = mcpy\phi(A, D_j^m)$ is redundant if $A_{\{m,p\}} \in AvailOut(D_j^m)$

Example of Redundancy Consider the Figure 4.4a and Figure 4.4b, in both these cases $B1$ and $B3$ have an $mcpy\phi$ operator, and there is no write to the array between this pair of $mcpy\phi$ statements. Thus, as Figure 4.5a shows, $A_{\{host,device\}}$ is available at the entry to basic block $B3$ which means the host and device memory space have the same copy of the array and any further copy is redundant. Thus we can remove the memory copy from the $B3$ in the first two cases.

Definition 5. ***Partial Redundancy****: A copy statement between memory spaces* m *and* p *for a particular array* A*, constitutes a partial redundancy, if both the memory spaces already have an updated copy on some but not all paths reaching the copy statement.*

Example of Partial Redundancy Next, consider the loop invariant case from Figure 4.4c. As Figure 4.5b shows, The memory copy of $B3$ is anticipable at the entry of both $B1$ and

$B2$, that is to the entry block of the loop. But the device definition in $B4$ makes sure that the $B5$ copy is not redundant. Now, the copy of $B5$ is available at the exit of $B5$ and also till the loop exit block $B7$. Consider the two edges of $B1 - B2$ and $B6 - B2$, $A_{\{host,device\}}$ is available only on the back edge $B6 - B2$, but not on the entry to the loop. Hence it is partially redundant at $B2$.

4.4.3 Lazy Code Motion

Partial redundancy elimination [76] eliminates redundant computation of expressions in programs by moving invariant computations out of loops and also eliminating identical computations that are performed more than once on any execution path. In this work, we use the formulation from the compiler textbook [74] and [75]. Our customized PRE algorithm for data movements has such 5 steps,

Step 1 *Basic block local properties*: compute the local properties of upward-exposed and downward-exposed $mcpy\phi$ operators using the transfer functions defined over the LASSA operators in Table 4.1.

Step 2 *Solve the data flow equations*: compute available and anticipable copy operations according to 2 and 3.

Step 3 *Determine Earliest and Latest placement*: given the solutions of availability and anticipability, we can determine the earliest point in the program at which it is safe to hoist the copy statement. It is profitable to insert a copy statement at a basic block *B*, if it makes other copy statements redundant. Again we use the original data flow equations[74], to solve for earliest and later placement.

Step 4 *Redundant copies*: this translates to identifying redundant memory copy statements according to 4.

Step 5 *Code rewrite*: identify the program point to insert the memory copy, and the set of redundant memory copies that can be deleted.

Note that the dataflow analysis on the LASSA IR ensures that the transformed program produces the same output as the original output. The semantics of the $mcpy\phi$ IR ensures the legality of the optimization.

4.5 LASSA Construction

This section introduces the location-aware heap SSA form, that is the intermediate representation for building the code analysis and transformation that optimizes the data movement. To clarify our setup, in this work,

- We are only dealing with arrays, and not considering other data structures like objects and linked lists.
- We assume each array is a non overlapping distinct

4.5.1 Why a new IR?

To enable the program analysis for heterogeneous computing, especially for memory access related dataflow analysis, we need a new intermediate representation to assist our dataflow analyzer. As mentioned in section 4.2, Heap SSA was initially designed for program analysis of arrays and object references and enabling optimizations like load/store elimination. Here we extend heap SSA to address the challenges mentioned in subsection 4.3.1. We want to develop an intermediate representation to address the following concerns regarding the memory concurrency model,

1. Distinguish access on different memory spaces
2. Express the relationship between two memory accesses from two concurrent tasks, running in the same memory space;
3. Express relationship between memory accesses on different memory spaces
4. Express communication between the different memory spaces

4.5.2 The Programming Model Assumptions

In this section we define the basic assumptions used to develop our analysis

- **Tasks** are sequence of instructions to be executed on a some computing resource. An abstract interpreter executes the instructions in a task in sequential order to ensure causality is not violated. Hence we assume tasks to be a single thread of execution.
- **Host Task** is a default task, that begins the program execution.
- **Device Task** are sequence of instructions launched on a target device. The host can launch several device tasks on one or more target devices, that can run asynchronously.
- **Memory Space** is the private persistent memory storage associated with each target task. Each memory space is labelled.
- **Data Copy** of named arrays can be initiated by the host, from one memory space to another.
- For any data copy, we are only concerned with the instructions, that are guaranteed to observe the effect of the copy, as specified by the underlying memory concurrency model.
- A program can have undefined behavior or data race, but any program transformation and optimization should still be valid as long as it adheres to the memory concurrency model.

To keep the discussions simple in this work, we mostly refer to named arrays, with non overlapping memory storage. But our representation is general enough to handle any heap storage, as in past work that showed how Heap SSA Form can be used for pointer analysis of programs that manipulate pointers to possibly overlapping objects in the heap.

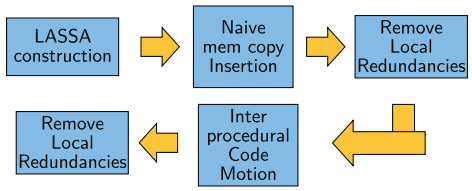


Figure 4.6: Code Optimization Framework

Problem Statement Given the above programming model, the optimizer first identifies where to insert the memory copy operations to ensure correctness. Then perform analysis to identify redundant memory copies, and finally perform code transformations to remove them. So, we build an intermediate representation to express the above programming model, and develop an analysis on that representation, to optimize the number of memory copies between different memory spaces. The Figure 4.6 shows the overall workflow of our optimization framework.

4.5.3 Auxiliary Analysis

The location-aware heap SSA (LASSA) presents the dataflow information for different memory spaces in a parallel execution environment. In this section we discuss the auxiliary analysis required to assist the LASSA construction and the subsequent dataflow analysis.

Happens-Before Analysis The happens-before relationship between two operations is one of the fundamental information for analyzing parallel program (e.g. the data-race analysis and the causality analysis). The complexity of the happens-before analysis depends on the underlying parallel programming model. The ambiguity of happens-before analysis is mainly from the control dependency, synchronization and pointer alias,

Pointer Alias Analysis To identify the redundancy in memory access and data synchronization, the analyzer needs to disambiguate the memory references precisely. As mentioned before, the pointer alias information also help on building happens-before analysis, thus we use flow-insensitive pointer analysis as the fundamental analysis that assists both happens-before analysis and memory reference disambiguation in this work.

```
1  main() {
2    int i, t;
3   // D_0^host = Init(A1)
4   // D_1^host = Init(B1)
5   float A1[N], B1[N];
6   // D_2^host = cφ(A1, D_0^host, D_3^host)
7   for (i=0; i < N; i++) {
8     // D_3^host = dφ(A1, D_2^host)
9     A1[i] = i;
10  }
11  // D_4^host = cφ(A1, D_3^host, D_10^host)
12  // D_5^host = cφ(B1, D_1^host, D_11^host)
13  for(t=1;t<=TSteps;t++) {
14    // D_6^host = uφ(A1, D_4^host)
15    // D_7^host = uφ(B1, D_5^host)
16    compute( A1, B1 );
17    // D_8^host = dφ(A1, D_6^host)
18    // D_9^host = dφ(B1, D_7^host)
19    if ( (t%2) == 0 ) {
20      // D_10^host = uφ(B1, D_9^host)
21      setStat( B1 );
22   } //D_11^host = cφ(B1, D_10^host, D_9^host)
23  }
24  // D_12^host = uφ(B1, D_11^host)
25  finalize(B1);
26  // D_13^host = dφ(B1, D_12^host)
27 }
28
29  void compute(float* Src,float *Dst){
30  // D_20^host = LiveOnEntry(Src)
31  // D_21^host = LiveOnEntry(Dst)
32  // D_22^0 = mcpyφ(Src, D_20^host)
33  // D_23^0 = mcpyφ(Dst, D_21^host)
34  #pragma omp target map(Src[0:N], Dst[0:N])
35  // D_24^0 = cφ(Src, D_22^0, D_26^0)
36  // D_25^0 = cφ(Dst, D_23^0, D_27^0)
37  for (int j = 0 ; j < N ; j++) {
38    // D_26^0 = uφ(Src, D_24^0)
39    // D_27^0 = dφ(Dst, D_25^0)
40    Dst[j] = Src[j]*j;
41  }
42  // D_28^host = mcpyφ(Src, D_26^0)
43  // D_29^host = mcpyφ(Dst, D_27^0)
44 }
```

Listing 4.5: LASSA Example embedded in LLVM IR

Memory Offset and Region Analysis A conservative analysis considers the write to an array A_i is reachable to every following read on array A_j that may alias to A_i later in the program. Regarding the imprecision due to array bounds analysis, the constant propagation is applied to identify the range of locations that an array expression read or write. Since the constant values do not always apply, it might not be possible to statically identify the range. Hence array definitions can not be killed. In sequential execution, a conservative analysis considers the write to an array A_i reachable to every read on array A_j later in the program, iff A_i and A_j may alias. We can refine the precision, by doing array index analysis, that can infer the range of locations that each array definition can write, or the array use can read. We reuse the past work on array index analysis, to improve the precision on heap SSA.

4.5.4 Location-Aware Heap SSA

Based on previous discussion, we extend the original heap SSA form, to *Location-Aware Heap SSA*(LASSA) form that incorporates the parallelism and distributed memory space information into heap SSA form. The two auxiliary analysis: pointer analysis and happens-before analysis should be applied before LASSA construction.

Operators

This LASSA is a look-aside information, which is constructed for the purpose of tracing memory accesses among different memory spaces. To uniquely identify each array access in the LASSA, we create a new version of the array for every corresponding access to the array. We define LASSA operators, that map an array version in one memory space, to another array version in the same or different memory space. We call these array versions as a definition. Since each LASSA operator creates a new definition, we can uniquely identify an LASSA operator by the corresponding definition.

We will use the following properties to define the LASSA operators.

Let D_x and D_y be two definitions/operators in the LASSA. We define a happens before

relationship between them, to represent the parallelism in the programming model.

Definition 6. ***Happens Before*** **($\prec$):** *If $D_x \prec D_y$, then the memory read/write operation D_y must be able to observe/overwrite the effect of D_x.*

Immediate Dominator($Idom$): of any definition refers to the most recent prevailing definition.

If $D_x\ Idom\ D_y$, then

- $D_x \prec D_y$, and
- $\nexists D_z \mid D_z \prec D_y \wedge D_x \prec D_z$, (There is no definition D_z that happens before D_y, but not D_x)

We use the notation, D_i^r, to denote a definition i in memory space r.

Definition 7. *We define the following operators in the LASSA:*

1. $D_i^r = d\phi(A, D_j^r)$ *creates a new "non-killing" definition D_i of an array A, such that, D_j^r is the prevailing definition of A just prior to D_i^r in same memory space r, that is, $D_j^r\ Idom_r\ D_i^r$.*
2. $D_k^r = c\phi(A, \{D_i^r, D_j^r\})$, *creates a control merge of the definitions $\{D_i^r, D_j^r\}$, for the array A;*
3. $D_i^r = u\phi(A, D_j^r)$, *denotes the read of array A, and* $D_j^r\ Idom_r\ D_i^r$
4. $D_i^r = mcpy\phi(A, D_j^p)$, *creates a new definition of array A, due to a copy from memory space p to memory space r; and $D_j^p \prec D_i^r$; similar to control merge, $mcpy\phi$ can merge multiple concurrent data copies: $D_i^r = mcpy\phi(A, D_j^p, D_k^q)$, and $D_j^p \prec D_i^r$, $D_k^q \prec D_i^r$.*

The semantics of the $d\phi$ and $u\phi$ operators are simply associated with the respective memory write and read operations. The $u\phi$ operator also generates a dummy definition, that is used only to infer the $\prec$ relationship with other LASSA operators. The functionality of control merge operator $c\phi$ is same as original heap SSA to define $\prec$ transitively. A $mcpy\phi$ is associated with a program point where the memory from source memory space data is actually flushed/written out to the destination memory space. *This guarantees the copied data is visible to any operations D_{dst} by the given pre-condition $mcpy\phi \prec D_{dst}$ in LASSA*. So, we can use $mcpy\phi$ for both synchronous or asynchronous memory copy, but the placement of the operator depends only on when the actual write is visible as defined by the memory concurrency model. The Listing 4.5 shows the corresponding LASSA for our motivating example.

Reaching Definitions

In this section, we describe the semantics of the LASSA, with respect to the reaching definition analysis for memory read and write.

Let $\mathbb{U}$ denote the set of all "non-killing" definitions $d\phi$ in the LASSA. Then reaching definitions is defined over a lattice of $\mathbb{U}$. That is, for any operation D_x^p in the LASSA, $\mathbb{RD}(D_x^p) \subseteq \mathbb{U}$.

Definition 8. ***Reaching Definition***, $\mathbb{RD}(D_x^p)$*:*

Let, D_x^p be a definition in our LASSA, then, $D_j^r \in \mathbb{RD}(D_x^p)$, if and only if,

1. $D_j^r \prec D_x^p$
2. $Alias(D_j^r, D_x^p) = true$

There are 4 kinds of operators in LASSA, we define the following transfer functions to update the $\mathbb{RD}$ at every operator.

- $\mathbb{RD}(D_i^r = d\phi(A, D_j^r)) = \mathbb{RD}(D_j^r) \cup D_i^r$

- $\mathbb{RD}(D_k^r = c\phi(A, \{D_i^r, D_j^r\})) = \mathbb{RD}(D_i^r) \cup \mathbb{RD}(D_j^r)$
- $\mathbb{RD}(D_i^r = u\phi(A, D_j^r)) = \mathbb{RD}(D_j^r)$
- $\mathbb{RD}(D_k^r = mcpy\phi(A, \{D_i^p, D_j^q\})) = \mathbb{RD}(D_i^p) \cup \mathbb{RD}(D_j^q)$

Table 4.2: Reaching Definitions for *compute* from Listing 4.5

LASSA op	$\mathbb{RD}$	LASSA op	$\mathbb{RD}$
D_{20}^{host}	$\{\}$	D_{25}^{0}	$\{D_{21}^{host}, D_{27}^{0}\}$
D_{21}^{host}	$\{\}$	D_{26}^{0}	$\{D_{20}^{host}\}$
D_{22}^{0}	$\{D_{20}^{host}\}$	D_{27}^{0}	$\{D_{21}^{host}, D_{27}^{0}\}$
D_{23}^{0}	$\{D_{21}^{host}\}$	D_{28}^{host}	$\{D_{20}^{host}\}$
D_{24}^{0}	$\{D_{20}^{host}\}$	D_{29}^{host}	$\{D_{21}^{host}, D_{27}^{0}\}$

The reaching definitions analysis, establishes the ground truth for our optimization. That is, any optimization we do must preserve the results of reaching definitions analysis. This analysis, computes, the set of definitions that must be reachable for a correct execution of the program.

The safety condition is that, we should not add or remove any reachable definition at any program point, after the optimization, to preserve the semantics of the original parallel program.

The Table 4.2 shows the reaching definitions for the function $compute$ from Listing 4.5.

4.5.5 Location-Aware heap SSA for OpenMP

As mentioned before, we use OpenMP as the underlying parallel programming model in this work, thus here we introduce how to build LASSA for OpenMP and the relevant issues for analysis.

Table 4.3: LASSA Operators for OpenMP target clauses

Clause	Inlined LASSA
target map tofrom	1 $D_i^1 = mcpy\phi(A, D_q^{host})$ 2 `#pragma omp target device(1) map(tofrom:A)` 3 `{ ... } //Device Code,` 4 $D_p^{host} = mcpy\phi(A, D_j^1)$
enter data alloc	1 $D_i^1 = Init(A)$ 2 `#pragma omp target device(1) enter data \` 3 `map(alloc:A)`
update data from	1 $D_i^{host} = mcpy\phi(A, D_j^1)$ 2 `#pragma omp target device(1) update data \` 3 `from(A)`

target wait	1 `#pragma omp target device(1) nowait`
	2 `depend(out:A) map(from:A)`
	3 `{ ... } // Device Code`
	4 `{ ... } // Host Code`
	5 $D_i^{host} = mcpy\phi(A, D_j^1)$
	6 `#pragma omp task depend(in:A)`

Note	D_j^1 is the most recent version of A on device 1

OpenMP Semantics

The OpenMP specification clearly defines the happens before relationship in terms of the **flush** operation, and lists the clauses that implicitly insert the **flush** operations. Based on this pre-condition, the happens before analysis for OpenMP program can be built. As LASSA is defined in terms of happens-before relationship, it can be constructed for presenting OpenMP programs.

In OpenMP, the cross device memory copy operations are mainly from the *target* clauses, including: **target**, **target data**, **target map**, **target enter/exit data**, and **target update**. There are also clauses that define the behaviors of copy operations, including **depend** and **nowait**. In the absence of **depend**, there is a default barrier, which manifests as a blocking memory copy. Hence a $mcpy\phi$ is inserted for every implicit/explicit blocking memory copy operations. **nowait** denotes an asynchronous data copy. The barrier is absent from the end of the target task, if **nowait** clause is specified. In that case, a $mcpy\phi$ is associated with

the program point of synchronization where the device memory is flushed and is visible to the host.

The Table 4.3 shows the LASSA corresponding to the fundamental OpenMP memory mapping clauses. These clauses cover most of the possible combinations [26] regarding the data movements.

LASSA Construction for OpenMP

Here we introduce the steps to build LASSA for a given structured parallel program, e.g., OpenMP. The basic algorithm is presented in Algorithm 3. The first two steps are the same as the original heap SSA, collecting all memory access (including memory copy) and building dominance information for them.

Algorithm 3: LASSA Construction for Structured Parallel Program

```
1  function LASSAConstruction ()
   Input  : OpenMP program P
   Output: OpenMP program P with look-aside information for LASSA
   //Collect memory operations
2  MemOps := CollectMemoryOps (P);
   //Build dominance information for each memory acccess and
     memory copy operations
3  dom := BuildDomInfo (P, MemOps);
4  pta := PointerAnalysis (P);
   //Task dependence analysis [77]
5  tdg := TaskDepAnalysis (P, pta);
6  foreach t ∈ tdg do
7      AssignMemSpaceID (t);
8  foreach op ∈ MemOps do
9      im := ImmDomInSameMemSpace (op); //Get immediate dominator
         from same memory space
10     F := GetCurrentFunc (op);
11     while ¬happensBefore (im, op) and im ≠ IsFunctionEntry (im) do
12         im := ImmDomInSameMemSpace (op);
13     ChainOps (im, op); //Chain the memory accesses as normal heap
         SSA does
14 Renaming (MemOps);
```

4.6 Evaluation

Implementation We implemented our analysis in LLVM 9.0.1 compiler framework [3]. The Figure 4.7 shows an overview of our analysis and optimization framework based on LLVM. We used *Clang* to emit LLVM IR and it also lowers the OpenMP directives to target-independent offload runtime library *libomptarget.so* APIs. The analysis pass then analyzes the API calls and their arguments to infer the offload pragmas specified by the user. We implemented an Andersen like flow-insensitive alias analysis, and also used the LLVM builtin analysis: scalar evolution for array index analysis and memory SSA [4] for chaining memory access and data copy operations.

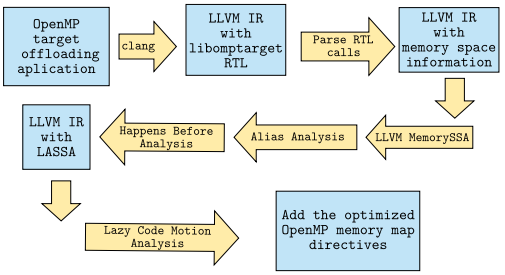
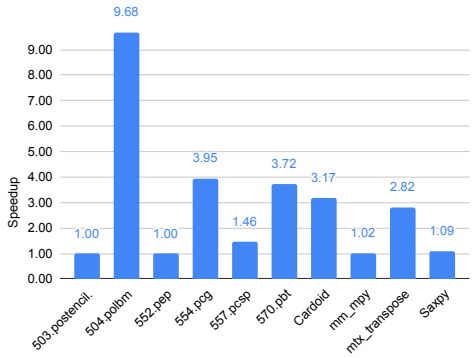


Figure 4.7: LLVM implementation of location-aware heap SSA and code optimization framework

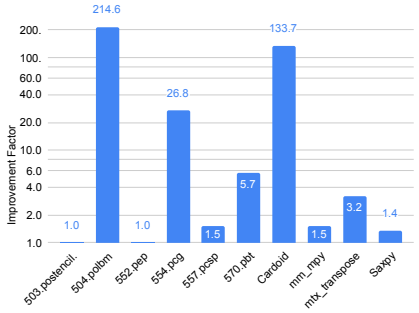
For optimal memory copy insertion, we built our analysis pass *OmpOptiMem* to perform an inter-procedural analysis that detects the redundant memory copies. Based on the analysis output we infer the optimal places to insert the OpenMP memory copy constructs. Then we use a perl script to parse output of *OmpOptiMem* and add the appropriate memory mapping directives to the original source file. Thus, given an OpenMP target offloading application with no explicit memory management, our tool analyzes the program and finally generates the modified source files after adding the optimal set of OpenMP memory map directives.

Experimental Setup We use Spec ACCEL v1.2 to evaluate our analysis and optimizations. It has 15 target offload benchmarks, but our implementation cannot handle bench-

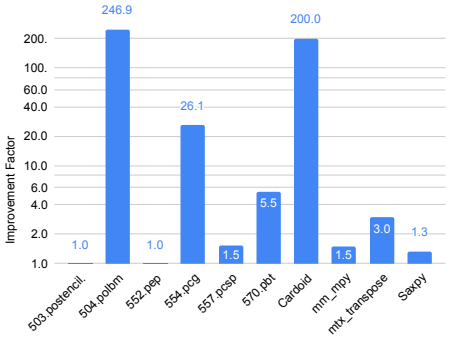
[3] http://llvm.org/
[4] https://llvm.org/docs/MemorySSA.html



(a) Speedup of our approach compared to using default mapping



(b) Improvement in Memory Copy Time



(c) Improvement in Total Bytes Copied

Benchmark	Input	Memory Copy Time	Total Time	Manual Speedup	Our Speedup
503.postencil	ref	954491.4	983668.3	33.5	1.0
503.postencil	test	3108.6	3205.2	25.6	1.0
503.postencil	train	3116.8	3211.5	25.5	1.0
504.polbm	ref	497859.3	553697.4	9.5	9.5
504.polbm	test	2014.5	2243.0	7.0	7.0
504.polbm	train	30222.4	33615.8	9.4	9.3
552.pep	ref	563182.7	671546.9	5.7	1.0
552.pep	test	469.8	653.9	3.6	1.0
552.pep	train	35889.8	42726.6	5.8	1.0
554.pcg	ref	807757.1	1040824.3	4.5	3.9
554.pcg	test	24129.3	31056.1	4.4	4.0
554.pcg	train	88261.0	113651.9	4.5	4.0
557.pcsp	ref	1204141.9	1308006.4	2.0	1.5
557.pcsp	test	20098.1	20229.1	1.5	1.5
557.pcsp	train	464849.7	475782.2	2.0	1.5
570.pbt	ref	3750608.5	4221773.7	3.7	3.7
570.pbt	test	1321807.1	1339861.0	2.6	2.6
570.pbt	train	2563893.7	2728456.2	3.6	3.6

(d) Comparison of Our achieved speedup with manually optimized speedup

Figure 4.8: Experimental Results

marks with "target declare" clause in LLVM, as it creates multiple IR modules, which can only be handled by a post link time optimization pass. Hence we show results on 6 SPEC benchmarks and include 4 other applications: *saxpy*, *Cardoid* ,*Matrix Multiply* and *Matrix Transpose*.

Our experiment results were gained from a linux workstation running Ubuntu 18.04.3 with Intel Core i5-7600 CPU (3.50GHz) with 16GB memory and a Nvidia "TITAN Xp" GPU with 12GB memory running CUDA 10.1.

Experimental Result and Discussion We removed all the explicit memory mapping

constructs specified in the benchmarks to obtain our baseline. Thus in our baseline version, every array is copied to device before launching the kernel and back to host after the kernel finishes.

After running our optimization on the benchmark we have 3 versions of each application: the baseline, *OmpOptiMem* optimized version, and the original hand optimized benchmark. We compare the performance of these three versions to evaluate our framework. We measure the efficiency on such metrics: the improvement of execution time, the reducing of data volumes and time consumed on data movement.

We did following study for the comparison with baseline code. The Figure 4.8a shows the overall speedup obtained by our approach compared the naive data mapping baseline. As we can see except *503* and *552*, all the benchmarks show a speedup ranging from $1.02\times$ to almost $10\times$. The 503 and 552 did not get chance to be optimized, due to the precision of alias analysis. The flow-insensitive pointer analysis could not disambiguate the array references in those two benchmarks. The Figure 4.8b explains the reason of the speedup, by showing the improvement factor of memory copy time, compared to the baseline. A significant point to note here is that the performance gain is largely dependent on the problem size (i.e. input data size), this also implies that the efficiency depends on the data volume reduced for transfer. Finally Figure 4.8c gives quantization study of the data volume transferred between host and device. It shows the reduction in total bytes copied. As is evident, there is a correlation between the factor by which total bytes were reduced and the obtained speedup. The speedup also depends on the pattern of computation. As the *Matrix Multiplication* example shows, even though there is a $1.5\times$ reduction in memory copy time, it does not translate to speedup, since the application is compute intensive. In the benchmark *Cardoid*, there is an outer loop which iterates for 100 iterations, and launches an inner loop on the target device. The default semantics of the **target** construct would copy in and out the arrays in each iteration. But, since there is no host access of the data, there is no need to copy the data back in and out every time. That is the

reason, we see almost 100% of the memory copies are eliminated after our optimization. Benchmark *Saxpy* is similar to *Cardoid*, there is an outer loop that launches the target task every iteration, and redundantly copies the data in every iteration. Figure 4.8d gives the comparison efficiency of our approach with user manually optimized code regarding SPEC ACCEL benchmarks on different input size. The 3rd and 4th columns give the memory copy time and total execution time for baseline code (i.e. naive memory mapping version). The 5th column shows the speedup obtained by comparing user manually optimized code against baseline. And the last column shows the speedup got from our approach. In general, the user manually optimized version gives the better improvement by comparing the last two columns, and our approach (i.e. compiler optimization) got similar performance on *504.polbm* and *570.pbt*. As mentioned above, there is no improvement from *503.postencil* and 552.pep due to the precision issues from pointer alias analysis. This study gave the evidence that the compiler can nearly generate code as efficient as the user's manually optimized version based on the dataflow analysis with enough precision.

4.7 Related Work

The problem of code generation and communication optimization for distributed memory machines is a classical problem, studied for a very long time. Amarasinghe and Lam [65] introduced a data flow analysis framework to first generate correct remote message read/write code, and then detect and remove redundancies. Chavarria and Mellor-Crummey [66] proposed a communication coalescing optimization to reduce redundant data transfer for High Performance Fortran applications. Dathathri et. al [69] elaborated polyhedral model to establish the static analysis and automatically generate efficient data movement code for non-shared address spaces. Load elimination and partial code motion are the classic optimization for eliminating the redundant memory load or computation in sequential program. In [78], Bodik et.al. phrased load-reuse problem as a path-sensitive analysis problem on dataflow graph. Their algorithm can detects the reuse pattern for both

scalar variable and pointer-based memory load operations. Barik and Sarkar [79] developed a load elimination technique for a structured parallel programming model: Habanero-Java. Kruse and Grosser developed DeLICM [80] , a code optimization that is targeted to eliminate problematic scalar dependences based on polyhedral value analysis.

Ramashekar and Boundhugula introduced BBMM[81]: an integrated compiler and runtime optimization system for tiling loop nests and running them on multi-GPU system. Their compiler applies communication optimization for the tiled loop nest and generated the OpenCL code that uses BBMM runtime API to perform buffer management and data communication.

Compared with past work, our approach is to establish a general compiler optimization framework that optimizes data movement across different memory spaces for heterogeneous computing, a problem that used to be handled as a runtime dependent optimization by the related works mentioned above. This framework reduces the data movement overheads and is applicable to parallel programming models that support heterogeneous computation.

4.8 Summary

The fast development of acceleration architectures and applications has made heterogeneous computing the norm for high-performance computing. The cost of high volume data movement to the accelerators is an important bottleneck both in terms of application performance and developer productivity. Memory management is still a manual task performed tediously by expert programmers. In this work, we develop a compiler analysis to automate memory management for heterogeneous computing. We propose an optimization framework that casts the problem of detection and removal of redundant data movements into a partial redundancy elimination (PRE) problem and applies the lazy code motion technique to optimize it. We chose OpenMP as the underlying parallel programming model and implemented our optimization framework in the LLVM toolchain. We evaluated it with ten benchmarks and obtained a geometric speedup of 2.3×, and reduced on average 50% of

the total bytes transferred between the host-GPU.

CHAPTER 5

DETECTING INCORRECT USAGE OF OPENMP DATA MAPPING PRAGMAS

5.1 Introduction

Open Multi-Processing (OpenMP) is a widely used directive-based parallel programming model that supports offloading computations from hosts to accelerator devices such as GPUs. It offers accelerator programming and supports heterogeneous computing systems with host CPUs and device accelerators (currently GPUs and FPGAs) from version 4.0 onward. Notable accelerator-related features in OpenMP include unstructured data mapping, asynchronous execution, and runtime routines for device memory management.

5.1.1 OMP Target offloading and Data mapping

OMP offers the `omp target` directive for offloading computations to devices and the `omp target data` directive for mapping data across the host and the corresponding device data environment. It is used to generate a target task that can be offloaded to a device, and also to map variables to the device data environment. The *omp target data* directive explicitly maps variables from a host environment to a device data environment. On heterogeneous systems, managing the movement of data between the host and the device can be challenging, and is often a major source of performance and correctness bugs. In the OpenMP accelerator model, hosts and devices have their own memory space – i.e., data environments – and data movement between device and host is supported either explicitly via the use of a `map` clause or, implicitly through default data-mapping rules. The optimal, or even correct, specification of map clauses can be non-trivial and error-prone because it requires users to reason about the complex dataflow analysis. To ensure that the map clauses are correct, the OpenMP programmers need to make sure that variables that

are defined in one data environments and used in another data environments are mapped accordingly across the different device and host data environments. Given a data map construct, its semantics depends on all the previous usages of the map construct. Therefore, dataflow analysis of map clauses is necessarily context-sensitive since the entire call sequence leading up to a specific map construct can impact its behavior.

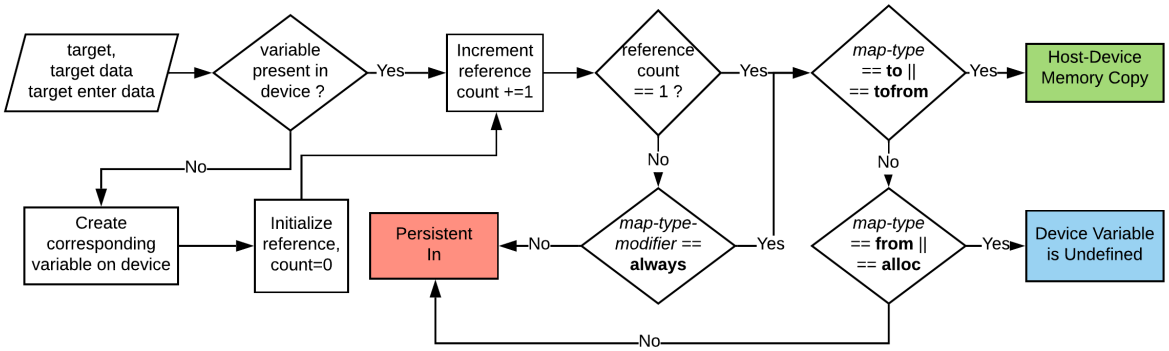
5.1.2 OpenMP 5.0 Map Semantics

Figure 5.1 shows a schematic illustration of the set of rules used when mapping a host variable to the corresponding list item in the device data environment, as specified in the OpenMP 5.0 standard. The rest of this work assumes that the accelerator device is a GPU, and that mapping a variable from host to device introduces a host-to-device memory copy, and vice-versa. However, the bugs that we identify reflect errors in the OpenMP code regardless of the target device.

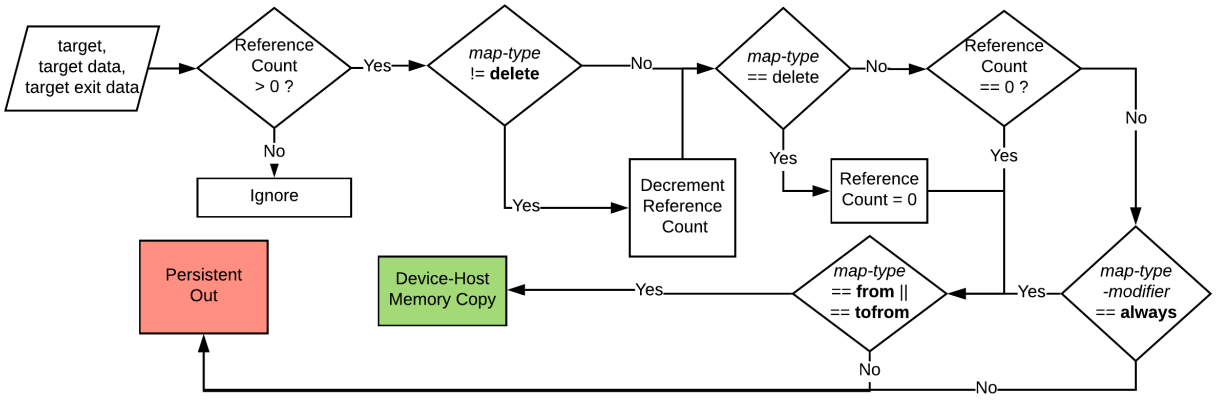
The different map types that OpenMP 5.0 supports are,

- `alloc`: allocate on device, uninitialized
- `to`: map to device before kernel execution, (host-device memory copy)
- `from`: map from device after kernel execution (device-host memory copy)
- `tofrom`: copy in and copy out the variable at the entry and exit of the device environment

Arrays are implicitly mapped as `tofrom`, while scalars are firstprivate in the target region implicitly, *i.e.,* the value of the scalar on the host is copied to the corresponding item on the device only at the entry to the device environment. As Figure 5.1 shows, OpenMP 5.0 specification uses the reference count of a variable, to decide when to introduce a device/host memory copy. The host to device memory copy is introduced only when the reference count is incremented from 0 to 1 and the `to` attribute is present. Then the reference count is incremented every time a new device map environment is created. The



(a) Flowchart for Enter Device Environment



(b) Flowchart for Exit Device Environment

Figure 5.1: Flowcharts to show how to interpret the map clause

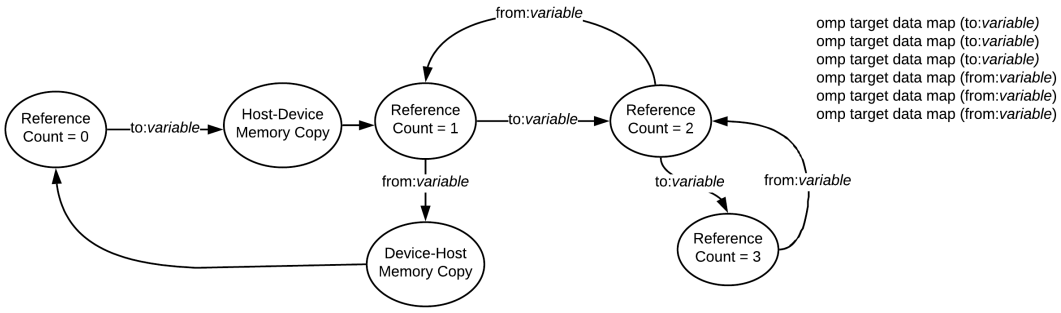


Figure 5.2: State Machine for inserting Host/Device Memory Copies

reference count is decremented on encountering a `from` or `release` attribute, while exiting the data environment. Finally, when the reference count is decremented to zero from 1, and the `from` attribute is present, the variable is mapped back to the host from the device. Figure 5.2 shows an example state machine, to decide when to insert the memory copies.

5.1.3 The Problem

For target offloading, the map clause is used to map variables from a task's data environment to the corresponding variable in the device data environment. Incorrect data map clauses can result in usage of stale data in either host or device data environment, which may result in the following kinds of issues,

- When reading the variable on the device data environment, it does not contain the updated value of its original variable.
- When reading the original variable, it was not updated with the latest value of the corresponding device environment variable.

5.1.4 Our Solution

We propose a static analysis tool called OMPSan to perform OpenMP code "sanitization". OMPSan is a compile-time tool, which statically verifies the correctness of the data mapping constructs based on a dataflow analysis. The key principle guiding our approach is

that: *an OpenMP program is expected to yield the same result when enabling or disabling OpenMP constructs*.

Our approach detects errors by comparing the dataflow information (reaching definitions via LLVM's memory SSA representation [82]) between the OpenMP and baseline code. We developed an LLVM-based implementation of our approach and evaluated its effectiveness using several case studies. Our specific contributions include:

- an algorithm to analyze OpenMP runtime library calls inserted by Clang in the LLVM IR, to infer the host/device memory copies. We expect that this algorithm will have applications beyond our OMPSan tool.
- a dataflow analysis to infer Memory def-use relations.
- a static analysis technique to validate if the host/device memory copies respect the original memory def-use relations.
- diagnostic information for users to understand how the map clause affects the host and device data environment.

Even though our algorithm is based on clang OpenMP implementation, it can very easily be applied to other approaches like using directives to delay the OpenMP lowering to a later LLVM pass.

This chapter is organized as follows. section 5.2 provides motivating examples to describe the common issues and difficulties in using OpenMP's *data map* construct. section 5.3 provides the background information that we use in our analysis. section 5.4 presents an overview of our approach to validate the usage of data mapping constructs. section 5.5 presents the LLVM implementation details, and section 5.6 presents the evaluation and some case studies. subsection 5.6.3 also lists some of the limitations of our tool, some of them common to any static analysis.

5.2 Motivating Examples

To motivate the utility and applicability of OMPSan, we discuss three potential errors in user code arising from improper usage of the data mapping constructs.

5.2.1 Default Scalar Mapping

```
1 int A[N], sum=0, i;
2 #pragma omp target
3 #pragma omp teams distribute parallel for reduction(+:sum) {
4     for(i=0; i<N; i++) {
5       sum += A[i];
6     }
7   }
8   printf("\n%d",sum);
```

Listing 5.1: Default scalar map

```
1 int A[N], sum=0, i;
2 #pragma omp target map(tofrom:sum)
3 #pragma omp teams distribute parallel for reduction(+:sum) {
4     for( i=0; i<N; i++) {
5       sum += A[i];
6     }
7   }
8   printf("\n%d",sum);
```

Listing 5.2: Explicit map

Consider the snippet of code in Listing 5.1. The `printf` on host, line 8, prints stale

value of sum. Note that the definition of sum on line 5 does not reach line 8, since the variable sum is not mapped explicitly using the `map` clause. As such, sum is implicitly `firstprivate`. As Listing 5.2 shows, an explicit `map` clause with the `tofrom` attribute is essential to specify the copy in and copy out of sum from device.

5.2.2 Reference Count Issues

Example 1:

```
1  int A[10], B[10];
2  for (int i =0 ; i < 10 ; i++)
3      A[i] = i;
4  #pragma omp target enter data map(to:A[0:10]) map(alloc:B[0:10])
5  #pragma omp target map(alloc:B[0:10])
6  for (int i = 0 ; i < 10; i++)
7      B[i] = A[i];
8  #pragma omp target exit data map(from:B[0:10])
9
10 for (int i = 0 ; i < 10; i++)
11     printf("%d",B[i]);
```

Listing 5.3: Usage of alloc

Listing 5.3 shows an example of data-mapping attributes across different data environments. The array B is specified as `alloc` in the first data environment. As per OpenMP 4.5 semantics (Figure 5.1), when exiting a data environment where a variable is mapped as `alloc`, there is no need to decrement the reference count. We can track the reference count for B is as follows,

- Line 5, reference count = 1

- Line 6, enter data environment, reference count = 2
- Line 8, exit data environment `alloc`, reference count = 2
- Line 9, exit data environment `from`, reference count =1

Note that a variable is mapped back from device to host only if its reference count is decremented to 0 upon exiting the device data environment. As such, on Line 12, the value of B is stale, since the updated value from the device was not mapped back to the host.

As Listing 5.4 shows, replacing `alloc` with `from` on line 6, will update the host version of B on exit of the map region at line 9. This is no longer a bug in OpenMP 5.0, since even `alloc` decrements the reference counter. This example just shows that certain nuances in the spec can to lead to incorrect behaviour.

```
1  int A[10], B[10];
2  for (int i =0 ; i < 10 ; i++)
3      A[i] = i;
4  #pragma omp target enter data map(to:A[0:10]) map(alloc:B[0:10])
5  #pragma omp target map(from:B[0:10])
6  for (int i = 0 ; i < 10; i++)
7      B[i] = A[i];
8  #pragma omp target exit data map(from:B[0:10])
9
10 for (int i = 0 ; i < 10; i++)
11     printf("%d",B[i]);
```

Listing 5.4: Usage of from

Example 2:

```
1  #define N 100
2  int A[N], sum=0;
3  #pragma omp target data map(from:A[0:N]) {
4  #pragma omp target map(from:A[0:N]) {
5      for(int i=0; i<N; i++) {
6          A[i]=i;
7        }
8      }
9      for(int i=0; i<N; i++) {
10       sum += A[i];
11     }
12   }
```

Listing 5.5: Reference Count

Listing 5.5 shows an example of a reference count issue.

The user declared the target data environment on line 3, with "A" mapped as "from". According to the OpenMP 4.5 semantics, the map clause on line 3, will instantiate an uninitialized version of array "A" on device, and also associate a reference count with it. The reference count will be set to 1, after the line 3. Now the map clause on the "target" construct, at line 4, will have no affect on the device copy of "A", but still it will increment the reference count to 2 at line 4. At the exit of the offloaded loop, after line 7, the reference count is 2, hence the "from" clause on line 4 will not have any affect, other than decrement the reference count to 1. Now the loop at line 9, that is executed on the host, will not be reading the updated version of "A" from the device, line 5, because "A" was not mapped back to the host after line 7. On exit of the data map region at line 12, the reference count is decremented to 0, and only then the device copy of "A" is mapped back and copied to the host. This is because of the "from" map attribute on line 3. In this example, "from"

attribute on line 4, has no affect. To fix this issue, we use the update clause as shown in Listing 5.6 to force the copy-out and to read the updated value of A on line 15.

This example shows the difficulty in interpreting an independent map construct. Especially when we are dealing with the global variables and map clauses across different functions, maybe even in different files, it becomes difficult to understand and identify potential incorrect usages of the map construct.

```
1
2  #define N 100
3  int A[N], sum=0;
4  #pragma omp target data map(from:A[0:N]) {
5  #pragma omp target map(from:A[0:N]) {
6        for(int i=0; i<N; i++) {
7          A[i]=i;
8        }
9      }
10 #pragma omp target update from(A[0:N])
11     for(int i=0; i<N; i++) {
12       sum += A[i];
13     }
14   }
```

Listing 5.6: Update Clause

5.3 Background

OMPSan assumes certain practical use cases, for example, in Listing 5.5, a user would expect the updated value of A on line 12. Having said that, a skilled ninja programmer may very well expect A to remain stale, because of their knowledge and understanding of the

complexities of data mapping rules. Our analysis and error/warning reports from this work are intended primarily for the former case.

5.3.1 Memory SSA form

Our analysis is based on the LLVM Memory SSA [82] [83], which is an imprecise implementation of Array SSA[84]. The Memory SSA is a virtual IR, that captures the def-use information for array variables. Every definition is identified by a unique name/number, which is then referenced by the corresponding use.

The Memory SSA IR has the following kinds of instructions/nodes,

- $INIT$, a special node to signify uninitialized or live on entry definitions
- $N' = MemoryDef(N)$, N' is an operation which may modify memory, and N identifies the last write that N' clobbers.
- $MemoryUse(N)$, is an operation that uses the memory written by the definition N, and does not modify the memory.
- $MemPhi(N_1, N_2, ...)$, is an operation associated with a basic block, and N_i is one of the may reaching definitions, that could flow into the basic block.

We make the following simplifying assumptions, to keep the analysis tractable

- Given an array variable we can find all the corresponding load and store instructions. So, we cannot handle cases, when pointer analysis fails to disambiguate the memory a pointer refers to.
- A $MemoryDef$ node clobbers the array associated with its store instruction. As a result, write to any array location, is considered to update the entire array.
- We analyze only the array variables that are mapped to a target region.

5.3.2 Scalar Evolution Analysis

LLVM's Scalar Evolution (SCEV) is a very powerful technique that can be used to analyze the change in the value of scalar variables over iterations of a loop. We can use the SCEV analysis to represent the loop induction variables as chain of recurrences. This mathematical representation can then be used to analyze the index expressions of the memory operations.

We implemented an analysis for array sections, that given a load/store, uses the LLVM SCEV analysis, to compute the minimum and maximum values of the corresponding index into the memory access. If the analysis fails, then we default to the maximum array size, which is either a static array, or can be extracted from the LLVM memory *alloc* instructions.

5.4 Our Approach

In this section, we outline the key steps of our approach with the algorithm and show a concrete example to illustrate the algorithm in action.

5.4.1 Algorithm

algorithm 5 shows an overview of our data map analysis algorithm. First, we collect all the array variables used in all the map clauses in the entire module. Then line 5, calls the function `ConstructArraySSA`, which constructs the Array SSA for each of the mapped Array variables. (In this work, we use "Array SSA" to refer to our extensions to LLVM's Memory SSA form by leveraging the capabilities of Array SSA form [84].) Then, we call the function, `InterpretTargetClauses`, which modifies the Array SSA graph, in accordance of the map semantics of the program. Then finally `ValidateDataMap` checks the reachability on the final graph, to validate the map clauses, and generates a diagnostic report with the warnings and errors.

Algorithm 1 Overview of Data Mapping Analysis

```
1: function DataMapAnalysis(Module)
2:     MappedArrayVars = φ
3:     for ArrayVar ∈ MapClauses do
4:         MappedArrayVars = MappedArrayVars ∪ ArrayVar
5:     ConstructArraySSA(Module, MappedArrayVars)
6:     InterpretTargetClauses(Module, MappedArrayVars)
7:     ValidateDataMap(MappedArrayVars)
8: function ConstructArraySSA(Module, MappedArrayVars)
9:     for MemoryAccess ∈ Module do
10:        ArrayVar = getArrayVar(MemoryAccess)
11:        if ArrayVar ∈ MappedArrayVars then
12:            if MemoryAccess ∈ OMP_targetOffload_Region then
13:                targetNode = true                    ▷ If Memory Access on device
14:            else
15:                targetNode = false                   ▷ If Memory Access on host
16:            Range = SCEVGetMinMax(MemoryAccess)
17:            underConstruction= GetArraySSA(ArrayVar)
18:                                                     ▷ could be null or incomplete
19:            InsertNodeArraySSA(underConstruction,MemoryAccess, targetNode, Range
   )
20:                                    ▷ Incrementally construct, by adding this access
21: function InterpretTargetClauses(Module, MappedArrayVars)
22:     for ArrayVar ∈ MappedArrayVars do
23:        ArraySSA = GetArraySSA(ArrayVar)
24:        for edge, (node, Successornode) ∈ (ArraySSA) do
25:            nodeIsTarget = isTargetOffload(node)
26:            succIsTarget = isTargetOffload(Successornode)
27:            if nodeIsTarget ! = succIsTarget then
28:                RemoveArraySSAEdge(node, Successornode )
29:    for dataMap ∈ dataMapClauses do
30:        hostNode = getHostNode(dataMap)
31:        deviceNode = getDeviceNode(dataMap)
32:        mapType = getMapClauseType(dataMap)
33:                                ▷ alloc/copyIn/copyOut/persistentIn/persistentOut
34:        InsertDataMapEdge(hostNode, deviceNode, mapType)
35: function ValidateDataMap(MappedArrayVars)
36:     for ArrayVar ∈ MappedArrayVars do
37:        ArraySSA = GetArraySSA(ArrayVar)
38:        for memUse ∈ getMemoryUseNodes(ArraySSA) do
39:            useRange = getReadRange(memUse)
40:            clobberingAccess = getClobberingAccess(ArraySSA, memUse)
41:            if isPartiallyReachable(ArraySSA, clobberingAccess, memUse, useRange)
   then
42:                Report WARNING
43:            else if isNotReachable(ArraySSA, clobberingAccess, memUse) then
44:                Report ERROR
```

Figure 5.3: Overview of Data Mapping Analysis

Algorithm 4: $computeTotalSpills$: Total Spills

```
   Data: EndBlock
   Result: TotalSpills
   //Loop's Exiting Basic Block :EndBlock
   //Assumption:  Loop has a single Exiting block
   //Total estimated spills after register allocation:  TotalSpills
1  Let, MaxLive[BB] = φ∀BB ;
2  Let, MaxRegistersReq = 0 ;
   //Assumption:  Nothing is live out from the loop body
   //Initialize live out from every Basic Block to null
3  Let, LiveOut[BB] = φ, ∀BB ;
4  Let, WorkQueue = {EndBlock} ;
5  while WorkQueue ≠ φ do
       //While the working Queue is not empty
6      Let, NextBB =Pop(WorkQueue) ;
       //If NextBB not already visited
7      Let, {LiveInBB, MaxLive[NextBB]} = getBBLiveSet(NextBB, LiveOut[NextBB]) ;
8      if MaxRegistersReq < |MaxLive[NextBB]| then
9          Let, MaxRegistersReq = |MaxLive[NextBB]| ;
10     foreach PredBB ∈ Predecessor(NextBB) do
11         Let, LiveOut[PredBB] = LiveOut[PredBB] ∪ LiveInBB ;
           //Live vars out of any Basic Block is the union of Live vars at the input
             of all its successors
12         Enqueue(WorkQueue, PredBB) ;
13 Let, TotalSpills = |MaxRegistersReq − AvailableHardwareRegisters| ;
```

Example

Let us consider the example in Figure 5.4a to illustrate our approach for analysis of data mapping clauses. `ConstructArraySSA` of algorithm 5, constructs the memory SSA form for arrays "A" and "C" as shown in Figure 5.4b. Then, `InterpretTargetClauses`, removes the edges between host and device nodes, as shown in Figure 5.4c, where the host is colored green and device is blue. Finally, the loop at line 29 of the function `InterpretTargetClauses`, introduces the host-device/device-host memory copy edges, as shown in Figure 5.4d. For example $L1$ is connected to $S2$ with a host-device memory copy for the enter data map pragma with `to`: $A[0:50]$ on line 5. Also, we connect the *INIT* node with $L2$, to account for the `alloc`:$C[0:100]$, which implies an uninitialized reaching definition for this example.

Lastly, `ValidateDataMap` function, traverses the graph, resulting in the following observations:

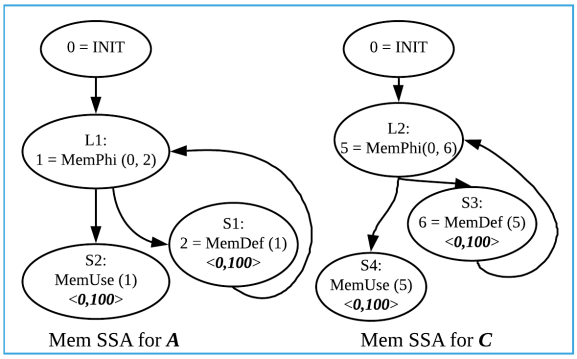
- (Error) Node $S4$:$MemUse(5)$ is not reachable from its corresponding definition $L2$:

Algorithm 5: Overview of Data Mapping Analysis

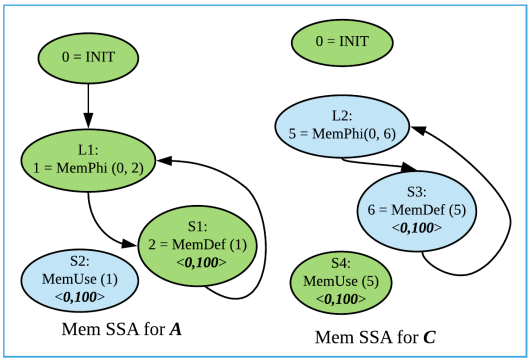
```
1  Function DataMapAnalysis(Module):
2      Let, MappedArrayVars = φ ;
3      foreach ArrayVar ∈ MapClauses do
4          Let, MappedArrayVars = MappedArrayVars ∪ ArrayVar ;
5      ConstructArraySSA(Module, MappedArrayVars) ;
6      InterpretTargetClauses(Module, MappedArrayVars) ;
7      ValidateDataMap(MappedArrayVars) ;
8  Function ConstructArraySSA(Module, MappedArrayVars):
9      foreach MemoryAccess ∈ Module do
10         Let, ArrayVar = getArrayVar(MemoryAccess) ;
11         if ArrayVar ∈ MappedArrayVars then
12             if MemoryAccess ∈ OMP_targetOffload_Region then
13                 Let, targetNode = true ;
                   //If Memory Access on device
14             else
15                 Let, targetNode = false //If Memory Access on host
16             Let, Range = SCEVGetMinMax(MemoryAccess) ;
17             Let, underConstruction= GetArraySSA(ArrayVar) ;
               //could be null or incomplete
18             InsertNodeArraySSA(underConstruction, MemoryAccess, targetNode, Range) ;
               //Incrementally construct, by adding this access
19 Function InterpretTargetClauses(Module, MappedArrayVars):
20     foreach ArrayVar ∈ MappedArrayVars do
21         Let, ArraySSA = GetArraySSA(ArrayVar) ;
22         foreach edge, (node, Successornode) ∈ (ArraySSA) do
23             Let, nodeIsTarget = isTargetOffload(node) ;
24             Let, succIsTarget = isTargetOffload(Successornode) ;
25             if nodeIsTarget != succIsTarget then
26                 Let, RemoveArraySSAEdge(node, Successornode ) ;
27     foreach dataMap ∈ dataMapClauses do
28         Let, hostNode = getHostNode(dataMap) ;
29         Let, deviceNode = getDeviceNode(dataMap) ;
30         Let, mapType = getMapClauseType(dataMap) ;
           //alloc/copyIn/copyOut/persistentIn/persistentOut
31         InsertDataMapEdge(hostNode, deviceNode, mapType) ;
32 Function ValidateDataMap(MappedArrayVars):
33     foreach ArrayVar ∈ MappedArrayVars do
34         Let, ArraySSA = GetArraySSA(ArrayVar) ;
35         foreach memUse ∈ getMemoryUseNodes(ArraySSA) do
36             Let, useRange = getReadRange(memUse) ;
37             Let, clobberingAccess = getClobberingAccess(ArraySSA, memUse) ;
38             if isPartiallyReachable(ArraySSA, clobberingAccess, memUse, useRange) then
39                 Report WARNING
40             else if isNotReachable(ArraySSA, clobberingAccess, memUse) then
41                 Report ERROR
```

```
1
2  L1: for ( i =0 ; i < 100 ; i++) {
3          S1: A[i] = 0;
4      }
5  #pragma omp target enter data map(
       to:A[0:50], alloc:C[0:100])
6  #pragma omp target
7  L2: for (i=0; i < 100; i++ ) {
8          S2: t = A[i] ;
9          S3 C[i] = t;
10     }
11 #pragma omp target exit data map(
       release:C[0:100])
12 L3: for (i=0; i < 100; i++) {
13         S4: print(C[i])
14     }
```

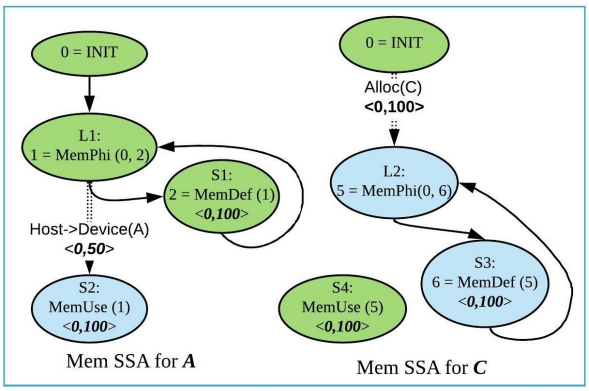
(a) Example, user Code



(b) Memory SSA for sequential version



(c) Classify Host/Device Regions (green=host, blue=device)



(d) Host/Device Memory Copies

Figure 5.4: Example of Data Map Analysis

$$5 = MemPhi(0, 6)$$

- (Warning) Only the partial artial array section $A[0:50]$, is reachable from definition $L1:1 = MemPhi(0,2)$ to $S2:MemUse(1)\langle 0:100\rangle$

section 5.6 contains other examples of the errors and warnings discovered by our tool.

5.5 Implementation

We implemented our framework in LLVM 8.0.0. The OpenMP constructs are lowered to runtime calls in Clang, so in the LLVM IR we only see calls to the OpenMP runtime. There are several limitations of this approach with respect to high level analysis like the one OMPSan is trying to accomplish. For example, the region of code that needs to be

offloaded to a device is opaque since it is moved to a separate function. These functions are in turn called from the OpenMP runtime library. As a result, it is challenging to perform a global data flow analysis for the memory def-use information of the offloaded region. To simplify the analysis, we have to compile with clang twice.

First, we compile the OpenMP program with the flag that enables parsing the OpenMP constructs, and compile it again without the flag, so that Clang ignores the OpenMP constructs and instead generates the baseline LLVM IR for the sequential version. During the OpenMP compilation pass, we execute our analysis pass, which parses the runtime library calls and generates a csv file that records all the user specified "target map" clauses, as explained in subsection 5.5.1.

Next we compile the program by ignoring the OpenMP pragmas, and perform whole program context and flow sensitive data flow analysis on LLVM code generated from the sequential version, to construct the Memory def-use chains, explained in subsection 5.5.2. Then this pass validates if the "target map" information recorded in the csv file, respects all the Memory def-use relations present in the sequential version of the code.

5.5.1 Interpreting OpenMP pragmas

The offload mechanism used by clang is to generate calls to a runtime library(RTL) whenever "target" directives are encountered. The offload library implements the routines shown in Table 5.1. In the LLVM IR, whenever we encounter a call to one of these RTL routines, we parse the arguments of the functions, and extract the relevant information from them. Table 5.2 lists the arguments that are relevant to interpret the semantics of the *map* clause.

```
1 #pragma omp target map(tofrom:A[0:10])
2     for (i = 0 ; i < 10; i++)
3       A[i] = i;
```

Listing 5.7: Example map clause

Table 5.1: Target Runtime Library Routines

RTL Routines	Arguments
$\_\_tgt\_target\_data\_begin$ :: Initiate a device data environment	`int64_t device_id, int32_t num_args void** args_base, void** args, int64_t *args_size, int64_t *args_maptype`
$\_\_tgt\_target\_data\_end$ :: Close a device data environment	`int64_t device_id, int32_t num_args void** args_base, void** args, int64_t *args_size, int64_t *args_maptype`
$\_\_tgt\_target\_data\_update$ :: Make a set of values consistent between host and device	`int64_t device_id, int32_t num_args void** args_base, void** args, int64_t *args_size, int64_t *args_maptype`
$\_\_tgt\_target$ :: Begin data environment, launch target region execution and end device environment	`int64_t device_id, void *host_addr, int32_t num_args void** args_base, void** args, int64_t *args_size, int64_t *args_maptype`
$\_\_tgt\_target\_teams$ :: Same as above, also specify number of teams and threads	`int64_t device_id, void *host_addr, int32_t num_args, void** args_base, void** args, int64_t *args_size, int64_t *args_maptype, int32_t num_teams, int32_t thread_limit`

Table 5.2: Target Runtime Library Routine Arguments Explanation

Argument	Explanation
$device\_id$	Uniquely Identify the target
num_args	Number of data pointers that require a mapping
$void** args$	Pointer to an array with num_args arguments, whose elements point to the first byte of the array section that needs to be mapped
$int64\_t* args\_size$	Pointer to an array with num_args arguments, whose elements contain the size in bytes of the array section to be mapped
$void** args\_base$	Pointer to an array with num_args arguments, whose elements point to base address and differs from $args$ if an array section does not start at 0
$void** args\_maptype$	Pointer to an array with num_args arguments, whose elements contain the required map atribute specified by the enum Table 5.3

Table 5.3: Target Runtime Library Map Type Attribute Enum

Enum Type	Map Clause
$OMP\_TGT\_MAPTYPE\_ALLOC$	alloc
$OMP\_TGT\_MAPTYPE\_TO$	to
$OMP\_TGT\_MAPTYPE\_FROM$	from
$OMP\_TGT\_MAPTYPE\_ALWAYS$	always
$OMP\_TGT\_MAPTYPE\_RELEASE$	release
$OMP\_TGT\_MAPTYPE\_DELETE$	delete
$OMP\_TGT\_MAPTYPE\_POINTER$	map a pointer instead of array

```
1  void **ArgsBase = {&A}
2  void **Args = {&A}
3  int64_t* ArgsSize = {40}
4  void **ArgsMapType = { OMP_TGT_MAPTYPE_TO | OMP_TGT_MAPTYPE_FROM }
5  call __tgt_target(-1, HostAdr, 1, ArgsBase,  Args, ArgsSize, ArgsMapType)
```

Listing 5.8: Pseudocode for LLVM IR with RTL calls

Listing 5.7 shows a very simple user program, with a target data map clause. Listing 5.8 shows the corresponding LLVM IR in pseudocode, after clang introduces the runtime calls at Line 5. We parse the arguments of this call to interpret the map construct. For example, the 3rd argument to the call at line 6 of Listing 5.8 is 1, that means there is only one item in the map clause. Line 1, that is the value loaded into $ArgsBase$ is used to get the memory variable that is being mapped. Line 3, $ArgsSize$ gives the end of the corresponding array section, starting from $ArgsBase$. Line 4, $ArgsMapType$, gives the map attribute used by the programmer, that is "tofrom".

We wrote an LLVM pass that analyzes every such Runtime Library (RTL) call, and tracks the value of each of its arguments, as explained above. Once we obtain this information, we use the algorithm in Figure 5.1 to interpret the data mapping semantics of each clause. The data mapping semantics can be classified into following categories,

- Copy In: A memory copy is introduced from the host to the corresponding list item in the device environment.
- Copy Out: A memory copy is introduced from the device to the host environment.
- Persistent Out: A device memory variable is not deleted, it is persistent on the device, and available to the subsequent device data environment.
- Persistent In: The memory variable is available on entry to the device data environment, from the last device invocation.

```
1  int main(){
2    int A[10], B[10];
3    for (int i =0 ; i < 10 ; i++)
4      A[i] = i;
5  #pragma omp target enter data map(to:A[0:10]) map(from:B[0:10])
6  #pragma omp target map(from:B[0:10])
7    for (int i = 0 ; i < 10; i++)
8      B[i] = A[i];
9  # pragma omp target data map(from:B[0:10],C[0:N])
10   for ( int i = 0 ; i < 10; i ++)
11     C [ i ] = B [ i ]*i;
12 #pragma omp target exit data map(from:B[0:10])
13   for (int i = 0 ; i < 10; i++)
14     printf("%d",B[i]);
15   return 0;
16 }
```

Listing 5.9: Example OpenMP map construct

The examples in subsection 5.6.2 illustrate the above classification. To illustrate the above classification, consider the example in Listing 5.9. Table 5.4 shows the data mapping inferred by our tool. For example "B" is persistent out of the first target region, that ends on line 9, and persistent in to the second target region on line 13. "B" is copy out only at the `exit data map` on line 13.

Line 6 of the example, creates a data environment, with "to" mapping for $A[0:10]$, while $B[0:10]$ is allocated on the device. This is illustrated in the first row of Table 5.4, the RTL signifies start of device data environment, line begin and end refer to the same line, with $A[0:10]$ as the copy in, and $B[0:10]$ as Alloc. Next target map clause on line 7,

Table 5.4: Output Data mapping for Listing 5.9

RTL name	Region Line Begin	Region Line end	Copy In	Persistent In	Copy Out	Persistent out	Alloc
$\_\_tgt\_target\_data\_begin$	6	6	A[0:10]				B[0:10]
$\_\_tgt\_target$	7	9		A[0:10], B[0:10]	A[0:10]	B[0:10]	
$\_\_tgt\_target$	10	12	A[0:10]	B[0:10]	A[0:10]	B[0:10]	
$\_\_tgt\_target\_data\_end$	13	13			B[0:10]		

specifies the offloaded region of code, along with a map clause. According to the OpenMP 4.5 semantics, as shown in the table, both $A[0:10]$ and $B[0:10]$, are persistent in, because of the "enter data map" clause on line 6. While $A[0:10]$ is copy out, $B[0:10]$ is still persistent out, and copied out only because of the "exit data map" clause on line 10.

5.5.2 Baseline Memory Use Def Analysis

LLVM has an analysis called the MemorySSA[82], it is a relatively cheap analysis that provides an SSA based form for memory def-use and use-def chains. LLVM MemorySSA is a virtual IR, which maps `Instructions` to `MemoryAccess`, which is one of three kinds, `MemoryPhi`, `MemoryUse` and `MemoryDef`.

Operands of any `MemoryAccess` are a version of the heap before that operation, and if the access can modify the heap, then it produces a value, which is the new version of the heap after the operation. Figure 5.5 shows the LLVM Memory SSA for the OpenMP program in Listing 5.10. The comments in the listing denote the LLVM IR and also the corresponding `MemoryAccess`.

We have simplified this example, to make it relevant to our context. `LiveonEntry` is a special `MemoryDef` that dominates every `MemoryAccess` within a function, and implies that the memory is either undefined or defined before the function begins. The first node in Figure 5.5 is a `LiveonEntry` node. The $3 = MemoryDef(2)$ node, denotes that there is a store instruction which clobbers the heap version 2, and generates heap 3, which represents the line 8 of the source code.

```
1  int main(){
2    int A[10], B[10];
3    // 2 = MemoryPhi(1,3)
4    for (int i =0 ; i < 10 ; i++) {
5      // %arrayidx = getelementptr %A, 0, %idxprom
6      // store %i.0, %arrayidx,
7      // 3 = MemoryDef(2)
8      A[i] = i;
9    }
10 #pragma omp target enter data map(to:A[0:5]) map(alloc:B[0:10])
11 #pragma omp target
12   // 4 = MemoryPhi(2,5)
13   for (int i = 0 ; i < 10; i++) {
14     // %arrayidx7 = getelementptr %A, 0, %idxprom6
15     // %2 = load %arrayidx7
16     // MemoryUse(4)
17     int t =  A[i];
18     // %arrayidx9 = getelementptr %B, 0, %idxprom8
19     // store %2, %arrayidx9
20     // 5 = MemoryDef(4)
21     B[i] = t
22   }
23
24   for (int i = 0 ; i < 10; i++) {
25     //arrayidx19 = getelementptr %B, 0, %idxprom18
26     //%3 = load %arrayidx19
27     // MemoryUse(4)
28     printf("%d",B[i]);
29   }
30   return 0;
31 }
```

Listing 5.10: OpenMP program, for Figure 5.5

Whenever more than one heap versions can reach a basic block, we need a $MemoryPhi$ node, for example, $2 = MemoryPhi(1, 3)$ corresponds to the for loop on line 4. There are two versions of the heap reaching this node, the heap 1, $1 = LiveonEntry$ and the other one from the back edge, heap 3, $3 = MemoryDef(2)$. The next `MemoryAccess`,

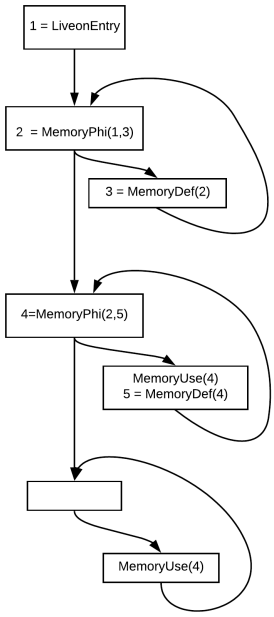


Figure 5.5: Memory SSA of Listing 5.10

$4 = MemoryPhi(2, 5)$, corresponds to the for loop at line 14. Again the clobbering accesses that reach it are 2 from the previous for loop and 5, from its loop body. The load of memory A on line 18, corresponds to the $MemoryUse(4)$, that notes that the last instruction that could clobber this read is `MemoryAccess` $4 = MemoryPhi(2, 5)$. Then, $5 = MemoryDef(4)$ clobbers the heap, to generate heap version 5. This corresponds to the write to array B on line 22. This is an important example of how LLVM deliberately trades off precision for speed. It considers the memory variables as disjoint partitions of the heap, but instead of trying to disambiguate aliasing, in this example, both stores/MemoryDefs clobber the same heap partition. Finally, the read of B on line 29, corresponds to $MemoryUse(4)$, with the heap version 4, reaching this load. Since this loop does not update memory, there is no need for a `MemoryPhi` node for this loop, but we have left the node empty in the graph to denote the loop entry basic block.

Now, we can see the difference between the LLVM memory SSA(Figure 5.5) and the array def-use chains required for our analysis(Figure 5.4). We developed a dataflow analysis to extract the array def-use chains from the LLVM Memory SSA, by disambiguating

```
28 int Mult(){
29
30     #pragma omp target map(to:a[0:C],b[0:C]) map(tofrom:c[0:C]) device(0)
31     {
32         #pragma omp teams distribute parallel for
33         for(int i=0; i<C; i++){
34             for(int j=0; j<C; j++){
35                 c[i]+=b[j+i*C]*a[j];
36             }
37         }
38     }
```

Listing 5.11: DRACC File 23

```
19 int init(){
20     for(int i=0; i<C; i++){
21         for(int j=0; j<C; j++){
22             b[j+i*C]=1;
23         }
24         a[i]=1;
25         c[i]=0;
26     }
~
31 int Mult(){
32
33     #pragma omp target map(to:a[0:C],b[0:C*C]) map(from:c[0:C*C]) device(0)
34     {
35         #pragma omp teams distribute parallel for
36         for(int i=0; i<C; i++){
37             for(int j=0; j<C; j++){
38                 c[i]+=b[j+i*C]*a[j];
```

Listing 5.12: DRACC File 30

the array variable that each load/store instruction refers to. So, for any store instruction, for example line 22, Listing 5.10, we can analyze the LLVM IR, and trace the value that the store instruction refers to, which is “B” as per the IR, comment of line 19.

We perform an analysis on the LLVM IR, which tracks the set of memory variables that each LLVM load/store instruction refers to. It is a context-sensitive and flow-sensitive iterative data flow analysis that associates each MemoryDef/MemoryUse with a set of memory variables. The result of this analysis is an array SSA form, for each array variable, to track its def-use chain, similar to the example in Figure 5.4.

5.6 Evaluation and Case Studies

For evaluating OMPSan we use the DRACC[85] suite, which is a benchmark for data race detection on accelerators, and also includes several data mapping errors also. Table 5.5 shows some distinct errors found by our tool in the benchmark[85] and the examples of

section 5.2. We were able to find the 15 known data mapping errors in the DRACC benchmark.

```
15 int init(){
16     for(int i=0; i<C; i++){
17         for(int j=0; j<C; j++){
18             b[j+i*C]=1;
19         }
20         a[i]=1;
21         c[i]=0;
22     }
23         return 0;
24 }
25
26
27 int Mult(){
28
29     #pragma omp target map(to:a[0:C]) map(tofrom:c[0:C]) map(alloc:b[0:C*C]) device(0)
30     {
31         #pragma omp teams distribute parallel for
32         for(int i=0; i<C; i++){
33             for(int j=0; j<C; j++){
34                 c[i]+=b[j+i*C]*a[j];
```

Listing 5.13: DRACC File 22

```
29     #pragma omp target enter data map(to:a[0:C],b[0:C*C],c[0:C]) device(0)
30     #pragma omp target device(0)
31     {
32         #pragma omp teams distribute parallel for
33         for(int i=0; i<C; i++){
34             for(int j=0; j<C; j++){
35                 c[i]+=b[j+i*C]*a[j];
36             }
37         }
38     }
39     #pragma omp target exit data map(release:c[0:C]) map(release:a[0:C],b[0:C*C]) device(0)
40     return 0;
41 }
42
43 int check(){
44     bool test = false;
45     for(int i=0; i<C; i++){
46         if(c[i]!=C){
```

Listing 5.14: DRACC File 26

Table 5.5: Errors found in the DRACC Benchmark and other examples

File Name	Error/Warning
DRACC File 22 Listing 5.13	ERROR Definition of :b on Line:18 is not reachable to Line:34, Missing Clause:to:Line:32
DRACC File 26 Listing 5.14	ERROR Definition of :c on Line:35 is not reachable to Line:46 Missing Clause:from/update:Line:44
DRACC File 30 Listing 5.12	ERROR Definition of :c on Line:25 is not reachable to Line:38 Missing Clause:to:Line:36
DRACC File 23 Listing 5.11	WARNING Line:30 maps partial data of :b smaller than its total size
Example in Listing 5.1	ERROR Definition of :sum on Line:5 is not reachable to Line:6 Missing Clause:from/update:Line:6
Example in Listing 5.5	ERROR Definition of :A on Line:7 is not reachable to Line:9 Missing Clause:from/update:8

Table 5.6: Time to Run OMPSan

Benchmark Name	-O3 Compilation Time (sec)	OMPSan Runtime (sec)
SPEC 504.polbm	17	16
SPEC 503.postencil	3	3
SPEC 552.pep	7	4
SPEC 554.pcg	15	9
NAS FT	32	15
NAS MG	34	31

5.6.1 Analysis Time

To get an idea of the runtime overhead of our tool, we also measured the runtime of the analysis. Table 5.6 shows the time to run OMPSan, on few SPEC ACCEL and NAS parallel benchmarks. Due to the context and flow sensitive data flow analysis implemented in OMPSan, its analysis time can be significant; however the analysis time is less than or equal to the *-O3* compilation time in all cases.

5.6.2 Diagnostic Information

Another major use case for OMPSan, is to help OpenMP developers understand the data mapping behavior of their source code. For example, Listing 5.15 shows a code fragment from the benchmark "FT" in the "NAS" suite. Our tool can generate the following information diagnostic information on the current version of the data mapping clause.

- $\_\_tgt\_target\_teams$, from::"ft.c:311" to "ft.c:331"
- Alloc: $u0\_imag[0:8421376], u0\_real[0:8421376]$
- Persistent In :: $twiddle[0:8421376], u1\_imag[0:8421376], u1\_real[0:8421376]$
- Persistent Out :: $twiddle[0:8421376], u0\_imag[0:8421376], u0\_real[0:8421376], u1\_imag[0:8421376], u1\_real[0:8421376]$
- Copy In:: *Null*, Copy Out:: *Null*

```
307 static void evolve(int d1, int d2, int d3)
308 {
309  int i, j, k;
311 #pragma omp target map (alloc: u0_real,u0_imag,u1_real,u1_imag,twiddle)
312  {
313 #pragma omp teams distribute
314   for (k = 0; k < d3; k++) {
315 #pragma omp parallel for
316    for (j = 0; j < d2; j++) {
317 #pragma omp simd
318     for (i = 0; i < d1; i++) {
319      u0_real[ ... ] = u0_real[ ... ]*twiddle[ ... ];
321      u0_imag[ ... ] = u0_imag[ ... ]*twiddle[ ... ];
```

Listing 5.15: *evolve* from NAS/ft.c

5.6.3 Limitations

Since OMPSan is a static analysis tool, it includes a few limitations.

- Supports statically and dynamically allocated array variables, but cannot handle dynamic data structures like linked lists It can possibly be addressed in future through advanced static analysis techniques (like shape analysis).

- Cannot handle target regions inside recursive functions. It can possibly be addressed in future work by improving our context sensitive analysis.

- Can only handle compile time constant array sections, and constant loop bounds. We can handle runtime expressions, by adding static analysis support to compare the equivalence of two symbolic expressions.

- Cannot handle "declare target" since it requires analysis across LLVM modules.

- May report false positives for irregular array accesses, like if a small section of the array is updated, our analysis may assume that the entire array was updated. More expensive analysis like symbolic analysis can be used to improve the precision of the static analysis.

- May fail if Clang/LLVM introduces bugs while lowering OpenMP pragmas to the RTL calls in the LLVM IR.

- May report false positives, if the OpenMP program relies on some dynamic reference count mechanism. Runtime debugging approach will be required to handle such cases.

It is interesting to note that, we did not find any false positives for the benchmarks we evaluated on.

5.7 Related Work

Managing data transfers to and from GPUs has always been an important problem for GPU programming. Several solutions have been proposed to help the programmer in managing

the data movement. CGCM [86] was one of the first systems with static analysis to manage CPU-GPU communications. It was followed by [67], a dynamic tool for automatic CPU-GPU data management. The OpenMPC compiler [20] also proposed a static analysis to insert data transfers automatically. [87] proposed a directive based approach for specifying CPU-GPU memory transfers, which included compile-time/runtime methods to verify the correctness of the directives and also identified opportunities for performance optimization. [68] proposed a compiler analysis to detect potential stale accesses and uses a runtime to initiate transfers as necessary, for the X10 compiler. [88] has also worked on automatically inferring the OpenMP mapping clauses using some static analysis. OpenMP has also defined standards, OMPT and OMPD[89, 90] which are APIs for performance and debugging tools. Archer[91] is another important work that combines static and dynamic techniques to identify data races in large OpenMP applications.

5.8 Summary

OpenMP offers directives for offloading computations from CPU hosts to accelerator devices such as GPUs. A key underlying challenge is in efficiently managing the movement of data across the host and the accelerator. User experiences have shown that memory management in OpenMP programs with offloading capabilities is non-trivial and error-prone.

This chapter presents **OmpSan** (OpenMP Sanitizer) – a static analysis-based tool that helps developers detect bugs from incorrect usage of the `map` clause, and also suggests potential fixes for the bugs. We have developed an LLVM based data flow analysis that validates if the def-use information of the array variables are respected by the mapping constructs in the OpenMP program. We evaluate **OmpSan** over some standard benchmarks and also show its effectiveness by detecting commonly reported bugs.

CHAPTER 6
CONCLUSION AND FUTURE WORK

In this dissertation, we address the problem of memory management for modern processors. Since memory is one of the most common systems performance bottlenecks, it is critical to understand the underlying memory architecture and tune it accordingly. We have observed that developers spend a lot of time and effort to manually implement various architecture-specific optimizations to improve their applications' memory performance.

We developed a static analysis to model three main aspects of the memory hierarchy. Firstly the data reuse at the cache, then GPU memory bandwidth utilization, and lastly, the host-device memory copy operations. We used the static analysis to guide compiler transformations that optimize either the data reuse or bandwidth utilization. We summarize our advancements and possible future directions in this chapter.

6.1 Static cache modeling to optimize data reuse via loop unrolling

In this work, we introduced OptiMemReuse, a new static analysis to guide the selection of unroll-jam transformations with the objective of reducing the memory cost of a loop nest while also taking register pressure into account. OptiMemReuse can be used to estimate the cache misses that would occur after performing the unroll-jam transformation for a given unroll configuration, without actually performing the transformation. We implemented OptiMemReuse in LLVM and compared the effect of unroll-jam driven by the OptiMemReuse cost model with the baseline register pressure model currently used in LLVM. The results obtained across all 30 PolyBench benchmarks are very encouraging. The performance improvements obtained by the use of OptiMemReuse were observed to be up to 2.57× for IBM POWER9 (geometric mean of 1.21×), up to 4.62× for AMD Ryzen (geometric mean of 1.33×), and up to 5.26× for Intel Xeon (geometric mean of 1.16×). These results sug-

gest that OptiMemReuse can be incorporated in any compiler that performs unroll-jam, so as to deliver significant performance improvements as a result.

6.1.1 Future work

One of the current limitations of our analysis is that it over-approximates triangular loops by its bounding rectangle. Our model also requires the static loop bounds, and assumes a constant in case the loop bounds are dynamic. It would be interesting to extend the analysis to handle more general loops. We can also extend the model by considering some profiled data as input to improve the accuracy of the estimated cache misses. Our memory cost model can also be applied to guide other loop transformations like loop distribution and loop fusion.

We can also extend the model to consider caches shared by multiple threads/cores and account for the interference between multiple cores and threads. This would make the model applicable to GPU L2 caches. This is possible only when the source code for the other threads is available to the compiler during static analysis.

6.2 Static GPU memory bandwidth modeling to improve bandwidth utilization via thread coarsening transformation

In this work, we proposed a compiler analysis approach to estimate the memory throughput achievable by a GPU kernel. We used the performance modeling strategy to guide a loop transformation to improve the memory bandwidth utilization. Our analysis does not depend on profiling data but is still able to achieve 96% of the ideal speedup on average. Furthermore, our compiler analysis is general enough and handles most common programming patterns.

We extended the OpenARC compiler to include the loop interleave transformation, and added the compiler analysis to predict the optimal interleave factor. The comprehensive experimental evaluation further validates that we can achieve the objective of our cost

function.

6.2.1 Future work

There are several enhancements that can be explored to generalize our optimization. The modeling framework can be extended to consider the effect of caches and shared memory. Currently, we assume all memory transactions miss in the caches and result in DRAM access.

The interleave transformation pass analyzes the high-level source code, which can be significantly modified by subsequent compiler passes. We observed slowdown in few kernels due to such transformations, like the introduction of Texture memory by OpenARC.

Currently, we only consider unrolling/interleaving along the innermost loop dimension, which corresponds to coarsening within a thread block. Our cost model can be extended to reason about the effect of interleaving along other dimensions.

Other interesting optimizations like vectorization can also take advantage of the loop interleaving by fusing the interleaved instructions. Task coarsening [92] is an analogous concept in OpenCL for FPGAs, we plan to extend our cost model to estimate the coarsening factor for FPGAs. We can also apply our cost model to other loop optimizations within OpenARC, like loop/kernel fusion.

6.3 Static modeling of host-GPU memory movement for optimization

In this work, we addressed an important problem regarding optimizing the data movement across different computation devices for heterogeneous computing. As most of the parallel programming language models (e.g., OpenMP, OpenACC) support offloading computations and data to different accelerators, automatically removing redundant memory copies to enhance the performance is a significant challenge for compiler construction. We developed an optimization framework to identify the redundant data movements and perform code translation to eliminate the redundancy. We first extended the Heap SSA to location-

aware heap SSA form (LASSA), an intermediate representation that can trace the memory access among different memory spaces from host to devices. Then we cast the problem of redundant memory copies to the partial redundancy elimination dataflow analysis on top of the LASSA. We evaluated our technique via 10 benchmarks written in OpenMP 4.5 with target offloading constructs. We achieved a Geomean speedup of 2.3X, and saved a Geomean 3480 MB of redundant data transfers. This tool will be open-sourced soon, for use by application developers.

6.3.1 Future work

We want to continue to develop our LASSA framework and apply it other use cases also. We specifically plan to do the following items

- Fix the existing issues with aliasing, to close the gap between explicit manual memory management benchmarks
- Extend our existing framework to overlap the compute with memory copy. This would utilize the asynchronous memory transfer features of OpenMP.
- We also want to incorporate the usage of unified memory, to handle the cases, when static analysis fails to determine the memory that is being used on device. As a fallback it can be dynamically loaded using unified memory
- We can also incorporate our static analysis with runtime approaces to handle cases when the static analysis fails.
- Since static analysis requires the source code to perform the analysis, we cannot handle libraries. We could rely on runtime approaches to handle OpenMP libraries also.
- Another direction worth exploring is to automatically manage the shared memory of GPUs using LASSA. We could model the GPU memory hierarchy to determine the

optimal usage of GPU memory hierarchy.

6.4 Detecting Incorrect usage of OpenMP data mapping pragmas

In this work, we have developed OMPSan, a static analysis tool to interpret the semantics of the OpenMP map clause, and deduce the data transfers introduced by the clause. Our algorithm tracks the reference count for individual variables to infer the effect of the data mapping clause on the host and device data environment. We have developed a data flow analysis, on top of LLVM memory SSA to capture the def-use information of Array variables. We use LLVM Scalar Evolution, to improve the precision of our analysis by estimating the range of locations accessed by a memory access. This enables the OMPSan to handle array sections also. Then OMPSan computes how the data mapping clauses modify the def-use chains of the baseline program, and use this information to validate if the data mapping in the OpenMP program respects the original def-use chains of the baseline sequential program. Finally OMPSan reports diagnostics, to help the developer debug and understand the usage of `map` clauses of their program. We believe the analysis presented in this work is very powerful and can be developed further for data mapping optimizations also. We also plan to combine our static analysis with a dynamic debugging tool, that would enhance the performance of the dynamic tool and also address the limitations of the static analysis.

REFERENCES

[1] D. A. Patterson and J. L. Hennessy, *Computer Architecture: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990, ISBN: 1558800698.

[2] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *SIGARCH Comput. Archit. News*, vol. 23, no. 1, 20–24, Mar. 1995.

[3] V. Sarkar, "Automatic selection of high-order transformations in the ibm xl fortran compilers," *IBM J. Res. Dev.*, vol. 41, no. 3, 233–264, May 1997.

[4] J. Ferrante, V. Sarkar, and W. Thrash, "On estimating and enhancing cache effectiveness," in *Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing*, Berlin, Heidelberg: Springer-Verlag, 1991, 328–343, ISBN: 354055422X.

[5] T. Gysi, T. Grosser, L. Brandner, and T. Hoefler, "A fast analytical model of fully associative caches," ser. PLDI 2019, Phoenix, AZ, USA: Association for Computing Machinery, 2019, 816–829, ISBN: 9781450367127.

[6] A. Agarwal, J. Hennessy, and M. Horowitz, "An analytical cache model," *ACM Trans. Comput. Syst.*, vol. 7, no. 2, 184–215, May 1989.

[7] M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm, "Cache behavior prediction by abstract interpretation," in *Static Analysis*, R. Cousot and D. A. Schmidt, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 52–66, ISBN: 978-3-540-70674-8.

[8] S. Ghosh, M. Martonosi, and S. Malik, "Cache miss equations: A compiler framework for analyzing and tuning memory behavior," *ACM Trans. Program. Lang. Syst.*, vol. 21, no. 4, 703–746, Jul. 1999.

[9] S. Chatterjee, E. Parker, P. J. Hanlon, and A. R. Lebeck, "Exact analysis of the cache behavior of nested loops," in *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, ser. PLDI '01, Snowbird, Utah, USA: Association for Computing Machinery, 2001, 286–297, ISBN: 1581134142.

[10] M. E. Wolf and M. S. Lam, "A data locality optimizing algorithm," *SIGPLAN Not.*, vol. 26, no. 6, 30–44, May 1991.

[11] K. Kennedy and K. S. McKinley, "Optimizing for parallelism and data locality," in *Proceedings of the 6th International Conference on Supercomputing*, ser. ICS '92, Washington, D. C., USA: Association for Computing Machinery, 1992, 323–334, ISBN: 0897914856.

[12] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '08, Tucson, AZ, USA: Association for Computing Machinery, 2008, 101–113, ISBN: 9781595938602.

[13] M. D. Lam, E. E. Rothberg, and M. E. Wolf, "The cache performance and optimizations of blocked algorithms," in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS IV, Santa Clara, California, USA: Association for Computing Machinery, 1991, 63–74, ISBN: 0897913809.

[14] K. K. Chang, *Understanding and improving the latency of dram-based memory systems*, 2017. arXiv: 1712.08304 [cs.AR].

[15] I. Gelado and M. Garland, "Throughput-oriented gpu memory allocation," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '19, Washington, District of Columbia: Association for Computing Machinery, 2019, 27–37, ISBN: 9781450362252.

[16] V. Volkov and J. Demmel, "Benchmarking gpus to tune dense linear algebra," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, IEEE Press, 2008, 1–11.

[17] V. Volkov, "Understanding latency hiding on gpus," PhD thesis, EECS Department, University of California, Berkeley, 2016.

[18] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the gpu," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '16, Barcelona, Spain: Association for Computing Machinery, 2016, ISBN: 9781450340922.

[19] X. Zhang, G. Tan, S. Xue, J. Li, K. Zhou, and M. Chen, "Understanding the gpu microarchitecture to achieve bare-metal performance tuning," *SIGPLAN Not.*, vol. 52, no. 8, 31–43, Jan. 2017.

[20] S. Lee and R. Eigenmann, "Openmpc: Extended openmp programming and tuning for gpus," in *Proceedings of the 2010 ACM/IEEE International Conference for High*

Performance Computing, Networking, Storage and Analysis, ser. SC '10, Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11, ISBN: 978-1-4244-7559-9.

[21] J. Wu, A. Belevich, E. Bendersky, M. Heffernan, C. Leary, J. Pienaar, B. Roune, R. Springer, X. Weng, and R. Hundt, "Gpucc: An open-source gpgpu compiler," in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, ser. CGO '16, Barcelona, Spain: Association for Computing Machinery, 2016, 105–116, ISBN: 9781450337786.

[22] A. Chandrasekhar, G. Chen, P.-Y. Chen, W.-Y. Chen, J. Gu, P. Guo, S. H. P. Kumar, G.-Y. Lueh, P. Mistry, W. Pan, T. Raoux, and K. Trifunovic, "Igc: The open source intel graphics compiler," in *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO 2019, Washington, DC, USA: IEEE Press, 2019, 254–265, ISBN: 9781728114361.

[23] H. Zhou, S. Bateni, and C. Liu, "Gru: Exploring computation and data redundancy via partial gpu computing result reuse," in *Proceedings of the 2018 International Conference on Supercomputing*, ser. ICS '18, Beijing, China: Association for Computing Machinery, 2018, 43–52, ISBN: 9781450357838.

[24] P. Barua, J. Shirako, and V. Sarkar, "Cost-driven thread coarsening for gpu kernels," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT 2018, Limassol, Cyprus: Association for Computing Machinery, 2018, ISBN: 9781450359863.

[25] P. Barua, J. Zhao, and V. Sarkar, "Ompmemopt: Optimized memory movement for heterogeneous computing," in *Euro-Par 2020: Parallel Processing*, M. Malawski and K. Rzadca, Eds., Cham: Springer International Publishing, 2020, pp. 200–216, ISBN: 978-3-030-57675-2.

[26] P. Barua, J. Shirako, W. Tsang, J. Paudel, W. Chen, and V. Sarkar, "Ompsan: Static verification of openmp's data mapping constructs," in *IWOMP*, 2019, ISBN: 978-3-030-28596-8.

[27] L.-N. Pouchet, *Polybench: The polyhedral benchmark suite*, 2015.

[28] S. Carr, K. S. McKinley, and C.-W. Tseng, "Compiler optimizations for improving data locality," *SIGOPS Oper. Syst. Rev.*, vol. 28, no. 5, 252–262, Nov. 1994.

[29] W. Bao, S. Krishnamoorthy, L.-N. Pouchet, and P. Sadayappan, "Analytical modeling of cache behavior for affine programs," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, Dec. 2017.

[30] F. E. Allen and J. Cocke, "A catalogue of optimizing transformations," 1972.

[31] V. Sarkar, "Optimized unrolling of nested loops," in *Proceedings of the 14th International Conference on Supercomputing*, ser. ICS '00, Santa Fe, New Mexico, USA: Association for Computing Machinery, 2000, 153–166, ISBN: 1581132700.

[32] S. Carr and Yiping Guan, "Unroll-and-jam using uniformly generated sets," in *Proceedings of 30th Annual International Symposium on Microarchitecture*, 1997, pp. 349–357.

[33] M. D. Hill, "Aspects of cache memory and instruction buffer performance," PhD thesis, EECS Department, University of California, Berkeley, 1987.

[34] S. Carr, K. S. McKinley, and C.-W. Tseng, "Compiler optimizations for improving data locality," in *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS VI, San Jose, California, USA: Association for Computing Machinery, 1994, 252–262, ISBN: 0897916603.

[35] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe, "Dependence graphs and compiler optimizations," in *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '81, Williamsburg, Virginia: Association for Computing Machinery, 1981, 207–218, ISBN: 089791029X.

[36] G. Goff, K. Kennedy, and C.-W. Tseng, "Practical dependence testing," in *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, ser. PLDI '91, Toronto, Ontario, Canada: Association for Computing Machinery, 1991, 15–29, ISBN: 0897914287.

[37] J. S. Harper, D. J. Kerbyson, and G. R. Nudd, "Analytical modeling of set-associative cache behavior," *IEEE Trans. Comput.*, vol. 48, no. 10, 1009–1024, Oct. 1999.

[38] J. Shirako, K. Sharma, N. Fauzia, L.-N. Pouchet, J. Ramanujam, P. Sadayappan, and V. Sarkar, "Analytical bounds for optimal tile size selection," in *Compiler Construction*, M. O'Boyle, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 101–121, ISBN: 978-3-642-28652-0.

[39] A. Qasem and K. Kennedy, "Profitable loop fusion and tiling using model-driven empirical search," in *Proceedings of the 20th Annual International Conference on Supercomputing*, ser. ICS '06, Cairns, Queensland, Australia: Association for Computing Machinery, 2006, 249–258, ISBN: 1595932828.

[40] H. Leather, M. O'Boyle, and B. Worton, "Raced profiles: Efficient selection of competing compiler optimizations," in *Proceedings of the 2009 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, ser. LCTES '09, Dublin, Ireland: Association for Computing Machinery, 2009, 50–59, ISBN: 9781605583563.

[41] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "A compiler framework for optimization of affine loop nests for gpgpus," in *Proceedings of the 22nd Annual International Conference on Supercomputing*, ser. ICS '08, Island of Kos, Greece: Association for Computing Machinery, 2008, 225–234, ISBN: 9781605581583.

[42] M. Stephenson and S. Amarasinghe, "Predicting unroll factors using supervised classification," in *International Symposium on Code Generation and Optimization*, 2005, pp. 123–134.

[43] u. Domagała, D. van Amstel, F. Rastello, and P. Sadayappan, "Register allocation and promotion through combined instruction scheduling and loop unrolling," in *Proceedings of the 25th International Conference on Compiler Construction*, ser. CC 2016, Barcelona, Spain: Association for Computing Machinery, 2016, 143–151, ISBN: 9781450342414.

[44] G. S. Murthy, M. Ravishankar, M. M. Baskaran, and P. Sadayappan, "Optimal loop unrolling for gpgpu programs," in *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, 2010, pp. 1–11.

[45] R. C. O. Rocha, V. Porpodas, P. Petoumenos, L. F. W. Góes, Z. Wang, M. Cole, and H. Leather, "Vectorization-aware loop unrolling with seed forwarding," in *Proceedings of the 29th International Conference on Compiler Construction*, ser. CC 2020, San Diego, CA, USA: Association for Computing Machinery, 2020, 1–13, ISBN: 9781450371209.

[46] NVIDIA, "Cuda programming guide," 2018.

[47] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *IEEE Des. Test*, vol. 12, no. 3, pp. 66–73, May 2010.

[48] (2018). Openacc, (visited on 07/29/2018).

[49] (2018). Openmp, (visited on 07/29/2018).

[50] J. Kim, S. Lee, and J. S. Vetter, "An openacc-based unified programming model for multi-accelerator systems," *SIGPLAN Not.*, vol. 50, no. 8, pp. 257–258, Jan. 2015.

[51] V. Sarkar, "Optimized unrolling of nested loops," in *Proceedings of the 14th International Conference on Supercomputing*, ser. ICS '00, Santa Fe, New Mexico, USA: ACM, 2000, pp. 153–166, ISBN: 1-58113-270-0.

[52] S. Unkule, C. Shaltz, and A. Qasem, "Automatic restructuring of gpu kernels for exploiting inter-thread data locality," in *Proceedings of the 21st International Con-*

ference on Compiler Construction, ser. CC'12, Tallinn, Estonia: Springer-Verlag, 2012, pp. 21–40, ISBN: 978-3-642-28651-3.

[53] A. Magni, C. Dubach, and M. O'Boyle, "Automatic optimization of thread-coarsening for graphics processors," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT '14, Edmonton, AB, Canada: ACM, 2014, pp. 455–466, ISBN: 978-1-4503-2809-8.

[54] A. Magni, C. Dubach, and M. F. P. O'Boyle, "A large-scale cross-architecture evaluation of thread-coarsening," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13, Denver, Colorado: ACM, 2013, 11:1–11:11, ISBN: 978-1-4503-2378-9.

[55] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather, "End-to-end deep learning of optimization heuristics," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2017, pp. 219–232.

[56] J. D. C. Little, "Or forum—little's law as viewed on its 50th anniversary," *Oper. Res.*, vol. 59, no. 3, pp. 536–549, May 2011.

[57] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, Jul. 1987.

[58] S. Lee and J. S. Vetter, "Openarc: Open accelerator research compiler for directive-based, efficient heterogeneous computing," in *HPDC Proceedings of the ACM Symposium on High-Performance Parallel and Distributed Computing, Short Paper*, 2014.

[59] H. Bae, D. Mustafa, J.-W. Lee, Aurangzeb, H. Lin, C. Dave, R. Eigenmann, and S. P. Midkiff, "The cetus source-to-source compiler infrastructure: Overview and evaluation," *Int. J. Parallel Program.*, vol. 41, no. 6, pp. 753–767, Dec. 2013.

[60] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, ser. IISWC '09, Washington, DC, USA: IEEE Computer Society, 2009, pp. 44–54, ISBN: 978-1-4244-5156-2.

[61] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The nas parallel benchmarks—summary and preliminary results," in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '91, Albuquerque, New Mexico, USA: ACM, 1991, pp. 158–165, ISBN: 0-89791-459-7.

[62] S. Lee and J. S. Vetter, "Openarc: Extensible openacc compiler framework for directive-based accelerator programming study," in *2014 First Workshop on Accelerator Programming using Directives*, 2014, pp. 1–11.

[63] S. Hong and H. Kim, "An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09, Austin, TX, USA: ACM, 2009, pp. 152–163, ISBN: 978-1-60558-526-0.

[64] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc, "A performance analysis framework for identifying potential benefits in gpgpu applications," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '12, New Orleans, Louisiana, USA: ACM, 2012, pp. 11–22, ISBN: 978-1-4503-1160-1.

[65] S. P. Amarasinghe and M. S. Lam, "Communication optimization and code generation for distributed memory machines," *SIGPLAN Not.*, vol. 28, no. 6, pp. 126–138, Jun. 1993.

[66] D. Chavarría-Miranda and J. Mellor-Crummey, "Effective communication coalescing for data-parallel applications," ser. PPoPP '05, Chicago, IL, USA: ACM, 2005, pp. 14–25, ISBN: 1-59593-080-9.

[67] T. B. Jablin, J. A. Jablin, P. Prabhu, F. Liu, and D. I. August, "Dynamically managed data for cpu-gpu architectures," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, ser. CGO '12, San Jose, California: ACM, 2012, pp. 165–174, ISBN: 978-1-4503-1206-6.

[68] S. Pai, R. Govindarajan, and M. J. Thazhuthaveetil, "Fast and efficient automatic memory management for gpus using compiler-assisted runtime coherence scheme," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '12, Minneapolis, Minnesota, USA: ACM, 2012, pp. 33–42, ISBN: 978-1-4503-1182-3.

[69] R. Dathathri, C. Reddy, T. Ramashekar, and U. Bondhugula, "Generating efficient data movement code for heterogeneous architectures with distributed-memory," ser. PACT '13, Edinburgh, Scotland, UK: IEEE Press, 2013, pp. 375–386, ISBN: 978-1-4799-1021-2.

[70] J. P. Hoeflinger and B. R. de Supinski, "The openmp memory model," in *OpenMP Shared Memory Parallel Programming*, Springer Berlin Heidelberg, 2008, ISBN: 978-3-540-68555-5.

[71] *Openmp*, Nov. 2019.

[72] S. J. Fink, K. Knobe, and V. Sarkar, "Unified analysis of array and object references in strongly typed languages," ser. SAS '00, London, UK, UK: Springer-Verlag, 2000, ISBN: 3-540-67668-6.

[73] K. Knobe and V. Sarkar, "Array ssa form and its use in parallelization," ser. POPL '98, San Diego, California, USA: ACM, 1998, pp. 107–120, ISBN: 0-89791-979-3.

[74] L. Torczon and K. Cooper, *Engineering A Compiler*, 2nd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007, ISBN: 012088478X.

[75] K.-H. Drechsler and M. P. Stadel, "A variation of knoop, rüthing, and steffen's lazy code motion," *SIGPLAN Not.*, vol. 28, no. 5, pp. 29–38, May 1993.

[76] J. Knoop, O. Rüthing, and B. Steffen, "Lazy code motion," ser. PLDI '92, San Francisco, California, USA: ACM, 1992, pp. 224–234, ISBN: 0-89791-475-9.

[77] V. Sarkar, "Compiler challenges for task-parallel languages," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming Language Design and Implementation*, ser. PLDI'11, San Jose, California, 2011.

[78] R. Bodík, R. Gupta, and M. L. Soffa, "Load-reuse analysis: Design and evaluation," ser. PLDI '99, Atlanta, Georgia, USA: ACM, 1999, pp. 64–76, ISBN: 1-58113-094-5.

[79] R. Barik and V. Sarkar, "Interprocedural load elimination for dynamic optimization of parallel programs," ser. PACT '09, Washington, DC, USA, 2009, pp. 41–52, ISBN: 978-0-7695-3771-9.

[80] M. Kruse and T. Grosser, "Delicm: Scalar dependence removal at zero memory cost," ser. CGO 2018, Vienna, Austria: ACM, 2018, pp. 241–253, ISBN: 978-1-4503-5617-6.

[81] T. Ramashekar and U. Bondhugula, "Automatic data allocation and buffer management for multi-gpu machines," *ACM Trans. Archit. Code Optim.*, vol. 10, no. 4, Dec. 2013.

[82] LLVM, "Llvm memoryssa,"

[83] D. Novillo, "Memory ssa- a unified approach for sparsely representing memory operations," in *Proceedings of the GCC Developers' Summit*, 2007.

[84] K. Knobe and V. Sarkar, "Array ssa form and its use in parallelization," in *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '98, San Diego, California, USA: ACM, 1998, pp. 107–120, ISBN: 0-89791-979-3.

[85] AachenUniversity, "Openmp benchmark,"

[86] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August, "Automatic cpu-gpu communication management and optimization," *SIGPLAN Not.*, vol. 46, no. 6, pp. 142–151, Jun. 2011.

[87] S. Lee, D. Li, and J. S. Vetter, "Interactive program debugging and optimization for directive-based, efficient gpu computing," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, 2014, pp. 481–490.

[88] G. Mendonça, B. Guimarães, P. Alves, M. Pereira, G. Araújo, and F. M. Q. a. Pereira, "Dawncc: Automatic annotation for data parallelism and offloading," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 2, 13:1–13:25, May 2017.

[89] A. Eichenberger, J. Mellor-Crummey, M. Schulz, N. Copty, J. DelSignore, R. Dietrich, X. Liu, E. Loh, and D. Lorenz, "Ompt and ompd: Openmp tools application programming interfaces for performance analysis and debugging," in *International Workshop on OpenMP (IWOMP 2013)*, 2013.

[90] A. E. Eichenberger, J. Mellor-Crummey, M. Schulz, M. Wong, N. Copty, R. Dietrich, X. Liu, E. Loh, and D. Lorenz, "Ompt: An openmp tools application programming interface for performance analysis," in *OpenMP in the Era of Low Power Devices and Accelerators*, A. P. Rendell, B. M. Chapman, and M. S. Müller, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 171–185, ISBN: 978-3-642-40698-0.

[91] S. Atzeni, G. Gopalakrishnan, Z. Rakamaric, D. H. Ahn, I. Laguna, M. Schulz, G. L. Lee, J. Protze, and M. S. Müller, "Archer: Effectively spotting data races in large openmp applications," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016, pp. 53–62.

[92] Q. Jia and H. Zhou, "Tuning stencil codes in opencl for fpgas," in *2016 IEEE 34th International Conference on Computer Design (ICCD)*, 2016, pp. 249–256.